Symfony RESTful API: Basics++ (Course 3)

With <3 from SymfonyCasts

Chapter 1: Pagination Design and Test

Tip

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

Hey, Guys! Welcome to Episode 3 of our REST in Symfony Series. In this episode, we're going to cover some really important details we haven't talked about yet, like pagination, filtering, and taking the serializer and doing really cool and custom things with it.

If you're following along with me, use the same project we've been building. If you're just joining, where have you been? Ah, it's fine: download the code from this page and move into the start/ directory. Start up the built-in PHP web server to get things running:

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

Designing how Pagination should Work

Let's talk about pagination first, because the /api/programmers endpoint doesn't have it. Eventually, once someone talks about our cool app on Reddit, we're going to have a lot of programmers here: too many to return all at once. First, think about pagination on the web. How does it work? Usually, it's done with query parameters: something like ?page=1, ?page=2, and so on. Sometimes, it's done in the URL - like /products/1 and /products/2. For API's, query parameters is better.

Second, on the web, we don't make the user guess those URLs: we give them links, like "next" and "previous", and maybe even "first" and "last".

So why would building an API be any different? Let's use query parameters and include links to help the API client get around.

Adding a Test

Like always, we're gonna start with a test because it's the easiest way to try things out *and* it helps us think about the API's design. In ProgrammerControllerTest, find the testProgrammersCollection() method and copy this to make a new test for pagination:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 88
    }
... lines 90 - 194
}
```

To make this interesting, we need more programmers - like 25. Add a for loop to do this: for i=0; i<25; i++. In each loop, create a programmer with the super creative name of Programmer plus the $i value. This means that we'll have programmers zero through 24. The avatarNumber is required, but we don't care about its value:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
        for ($i = 0; $i < 25; $i++) {
            $this->createProgrammer(array(
                'nickname' => 'Programmer'.$i,
                'avatarNumber' => 3,
            ));
        }

        // page 1
... lines 82 - 88
    }
... lines 90 - 194
}
```

Keep the same URL and the 200 status code assertion. Below, start basic with a sanity check for page 1: assert that the programmer with index 5 is equal to Programmer5:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 80
        // page 1
        $response = $this->client->get('/api/programmers');
        $this->assertEquals(200, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            'programmers[5].nickname',
            'Programmer5'
        );
    }
... lines 90 - 194
}
```

I'll use multiple lines to keep things clear. Index 5 is actually the 6th programmer, but since we start with Programmer0, this should definitely be Programmer5.

Adding count and total

It might also be useful to tell the API client how many results are on *this* page and how many results there are in total. I want to show 10 results per page in the API so add a line that looks for a new property called count that's set to 10. Let's also have another property called total. That'll be the *total* number of results. In this case, that should be 25:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 89
        $this->asserter()->assertResponsePropertyEquals($response, 'count', 10);
        $this->asserter()->assertResponsePropertyEquals($response, 'total', 25);
... line 92
    }
... lines 94 - 198
}
```

Adding Links

Finally, the API response needs to have those links! And by "links", I mean that I want to add a new field - maybe called "next" - whose value will be the URL to get the next page of results. Use the asserter again and change this to assertResponsePropertyExists(). Let's assert that there is an _links.next key, which means the JSON will have an _links key and a next key under that:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 91
        $this->asserter()->assertResponsePropertyExists($response, '_links.next');
    }
... lines 94 - 198
}
```

By moving things under _links, it makes it a little more obvious that next isn't a property of a programmer, but something different: a link.

Oh, and you probably saw my mistake above: change the line above to total, not count.

Following Links

And here's where things get really cool. In our test, we need to make a request to page 2 and make sure we see the next 10 programmers. Instead of hardcoding the URL, we can *read* the next link and use that for the next request. It's like the API version of clicking links!

Use $this->asserter() and then a method called readResponseProperty() to read the _links.next property. Now, add $response = $this->client->get($nextUrl) to go to the next page:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 93
        // page 2
        $nextLink = $this->asserter()->readResponseProperty($response, '_links.next');
        $response = $this->client->get($nextLink);
... lines 97 - 103
    }
... lines 105 - 209
}
```

Ok, let's test page 2! Copy some of the asserts that we just wrote. This time, the programmer with index 5 should be Programmer15 because we're looking at results 11 through 20. Next, the count should still be 10, and the total still 25 - but let's save a little code and remove that line:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 96
        $this->assertEquals(200, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            'programmers[5].nickname',
            'Programmer15'
        );
        $this->asserter()->assertResponsePropertyEquals($response, 'count', 10);
    }
... lines 105 - 209
}
```

The next link is nice. But we can do even more by *also* having a first link, a last link and a prev link unless we're on page 1. Copy the code from earlier that clicked the next link. Ooh, and let me fixing my formatting!

This time, use the _links.last key and update the variable to be $lastUrl. When we make a request to the final page, programmers[4] will be the last programmer because we started with index 0. The name should be Programmer24. And on this last page, count should be just 5:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 104
        $lastLink = $this->asserter()->readResponseProperty($response, '_links.last');
        $response = $this->client->get($lastLink);
        $this->assertEquals(200, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            'programmers[4].nickname',
            'Programmer24'
        );

... line 114
        $this->asserter()->assertResponsePropertyEquals($response, 'count', 5);
    }
... lines 117 - 221
}
```

I'm also going to use the asserter with assertResponsePropertyDoesNotExist() to make sure that there is *no* programmer here with index 5. Specifically, check for no programmers[5].nickname path:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
... lines 74 - 113
        $this->asserter()->assertResponsePropertyDoesNotExist($response, 'programmers[5].name');
        $this->asserter()->assertResponsePropertyEquals($response, 'count', 5);
    }
... lines 117 - 221
}
```

There's a small bug in my asserter code: if I just check for programmers[5], it thinks it exists but is set to null. That's why I'm checking for the nickname key.

That's it! Our pagination system is now *really* well-defined. Next, we'll bring this all to life.

Chapter 2: Pagerfanta Pagination

Installing Pagerfanta

To handle pagination, we're going to install the WhiteOctoberPagerfantaBundle. To install the bundle, run:

```
$ composer require white-october/pagerfanta-bundle
```

Pagerfanta is a great library for pagination, whether you're doing things on the web or building an API. While we're waiting, enable the bundle in AppKernel:

41 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 20
            new WhiteOctober\PagerfantaBundle\WhiteOctoberPagerfantaBundle(),
        );
... lines 23 - 33
    }
... lines 35 - 39
}
```

And that's it for setup: no configuration needed. Now just wait for Composer, and we're ready!

Setting up the Query Builder

Open up ProgrammerController and find the listAction() that we need to work on. Pagination is pretty easy: you basically need to tell the pagination library what page you're on and give it a query builder. Then, you can use it to fetch the correct results for that page.

To read the page query parameter, type-hint the Request argument and say $page = $request->query->get('page', 1);. The 1 is the default value in case there is *no* query parameter:

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

```
... lines 1 - 20
class ProgrammerController extends BaseController
{
... lines 23 - 74
    /**
     * @Route("/api/programmers")
     * @Method("GET")
     */
    public function listAction(Request $request)
    {
        $page = $request->query->get('page', 1);
... lines 82 - 102
    }
... lines 104 - 202
}
```

Go Deeper!

You could also use $request->query->getInt('page', 1) instead of get() to convert the page query parameter into an integer. See accessing request data for other useful methods.

Next, replace $programmers with $qb, standing for query builder. And instead of calling findAll(), use a new method called findAllQueryBuilder():

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

```
... lines 1 - 20
class ProgrammerController extends BaseController
{
... lines 23 - 78
    public function listAction(Request $request)
    {
        $page = $request->query->get('page', 1);

        $qb = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findAllQueryBuilder();
... lines 86 - 102
    }
... lines 104 - 202
}
```

That doesn't exist yet, so let's go add it!

I'll hold cmd and click to go into the ProgrammerRepository. Add the new method: public function findAllQueryBuilder(). For now, just return $this->createQueryBuilder(); with an alias of programmer:

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

```
... lines 1 - 8
class ProgrammerRepository extends EntityRepository
{
... lines 11 - 28
    public function findAllQueryBuilder()
    {
        return $this->createQueryBuilder('programmer');
    }
}
```

Perfect!

Creating the Pagerfanta Objects

This is *all* we need to use Pagerfanta. In the controller, start with $adapter = new DoctrineORMAdapter() - since we're using Doctrine - and pass it the query builder. Next, create a $pagerfanta variable set to new Pagerfanta() and pass *it* the adapter.

On the Pagerfanta object, call setMaxPerPage() and pass it 10. And then call $pagerfanta->setCurrentPage() and pass it $page:

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

```
... lines 1 - 10
use Pagerfanta\Adapter\DoctrineORMAdapter;
use Pagerfanta\Pagerfanta;
... lines 13 - 20
class ProgrammerController extends BaseController
{
... lines 23 - 78
    public function listAction(Request $request)
    {
... lines 81 - 85
        $adapter = new DoctrineORMAdapter($qb);
        $pagerfanta = new Pagerfanta($adapter);
        $pagerfanta->setMaxPerPage(10);
        $pagerfanta->setCurrentPage($page);
... lines 90 - 102
    }
... lines 104 - 202
}
```

Using Pagerfanta to Fetch Results

Ultimately, we need Pagerfanta to return the programmers that should be showing *right now* based on whatever page is being requested. To get that, use $pagerfanta->getCurrentPageResults(). But there's a problem: instead of returning an array of Programmer objects, this returns a type of traversable object with those programmes inside. This confuses the serializer. To fix that, create a new programmers array: $programmers = [].

Next, loop over that traversable object from Pagerfanta and push each Programmer object into our simple array. This gives us a clean array of Programmer objects:

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

```
... lines 1 - 20
class ProgrammerController extends BaseController
{
... lines 23 - 78
    public function listAction(Request $request)
    {
... lines 81 - 90
        $programmers = [];
        foreach ($pagerfanta->getCurrentPageResults() as $result) {
            $programmers[] = $result;
        }
... lines 95 - 102
    }
... lines 104 - 202
}
```

And that means we're dangerous. In createApiResponse, we *still* need to pass in the programmers key, but we also need to add count and total. Add the total key and set it to $pagerfanta->getNbResults().

For count, that's easy: that's the current number of results that are shown on *this* page. Just use count($programmers):

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

```
... lines 1 - 20
class ProgrammerController extends BaseController
{
... lines 23 - 78
    public function listAction(Request $request)
    {
... lines 81 - 95
        $response = $this->createApiResponse([
            'total' => $pagerfanta->getNbResults(),
            'count' => count($programmers),
            'programmers' => $programmers,
        ], 200);

        return $response;
    }
... lines 104 - 202
}
```

We're definitely not done, but this should be enough to return a valid response on page 1 at least. Test it out. Copy the method name and use --filter to *just* run that test:

```
$ ./bin/phpunit -c app --filter testGETProgrammersCollectionPaginated
```

This fails. But look closely: we *do* have programmers 0 through 9 in the response for page 1. It fails when trying to read the _links.next property because we haven't added those yet.

The PaginatedCollection

Before we add those, there's one improvement I want to make. Since we'll use pagination in a lot of places, we're going to need to duplicate this JSON structure. Why not create an object with these properties, and then let the serializer turn *that* object into JSON?

Create a new directory called Pagination. And inside of that, a new class to model this called PaginatedCollection. Make sure it's in the AppBundle\Pagination namespace. Very simply: give this 3 properties: items, total and count:

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

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

class PaginatedCollection
{
    private $items;

    private $total;

    private $count;
... lines 12 - 18
}
```

Generate the constructor and allow items and total to be passed. We don't need the count because again we can set it with $this->count = count($items). That should do it!

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

```
... lines 1 - 4
class PaginatedCollection
{
... lines 7 - 12
    public function __construct(array $items, $totalItems)
    {
        $this->items = $items;
        $this->total = $totalItems;
        $this->count = count($items);
    }
}
```

But something *did* just change: this object has an items property instead of programmers. That will change the JSON response. I made this change because I want to re-use this class for other resources. With a little serializer magic, you *could* make this dynamic: programmers in this case and something else like battles in other situations. But instead, I'm going to stay with items. This is something you often see with APIs: if they have their collection results under a key, they often use the same key - like items - for all responses.

But this means that I just changed our API. In the test, search for programmers: all of these keys need to change to items, so make sure you find them all:

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

```
... lines 1 - 5
6   class ProgrammerControllerTest extends ApiTestCase
7   {
... lines 8 - 53
54      public function testGETProgrammersCollection()
55      {
... lines 56 - 66
67          $this->asserter()->assertResponsePropertyIsArray($response, 'items');
68          $this->asserter()->assertResponsePropertyCount($response, 'items', 2);
69          $this->asserter()->assertResponsePropertyEquals($response, 'items[1].nickname', 'CowboyCoder');
70      }
... line 71
72      public function testGETProgrammersCollectionPaginated()
73      {
... lines 74 - 83
84          $this->asserter()->assertResponsePropertyEquals(
85              $response,
86              'items[5].nickname',
87              'Programmer5'
88          );
... lines 89 - 97
98          $this->asserter()->assertResponsePropertyEquals(
99              $response,
100             'items[5].nickname',
101             'Programmer15'
102         );
... lines 103 - 107
108         $this->asserter()->assertResponsePropertyEquals(
109             $response,
110             'items[4].nickname',
111             'Programmer24'
112         );
113
114         $this->asserter()->assertResponsePropertyDoesNotExist($response, 'items[5].name');
... line 115
116     }
... lines 117 - 221
222 }
```

Using the new class is easy: $paginatedCollection = new PaginatedCollection(). Pass it $programmers and $pagerfanta->getNbResults().

To create the ApiResponse pass it the $paginatedCollection variable directly: $response = $this->createApiResponse($paginatedCollection):

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

```
... lines 1 - 10
use AppBundle\Pagination\PaginatedCollection;
... lines 12 - 21
class ProgrammerController extends BaseController
{
... lines 24 - 79
    public function listAction(Request $request)
    {
... lines 82 - 96
        $paginatedCollection = new PaginatedCollection($programmers, $pagerfanta->getNbResults());
... lines 98 - 117
        $response = $this->createApiResponse($paginatedCollection, 200);

        return $response;
    }
... lines 122 - 220
}
```

Try the test!

```
$ ./bin/phpunit -c app --filter testGETProgrammersCollectionPaginated
```

It still fails, but only once it looks for the links. The first response looks exactly how we want it to. Okay, that's awesome - so now let's add some links.

Chapter 3: Pagination Links

The response *is* returning a paginated list, and it even has extra count and total fields. Now we need to add those next, previous, first and last links. And since the response is entirely created via this PaginatedCollection class, this is simple: just add a new private $_links = array(); property:

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

```
... lines 1 - 4
class PaginatedCollection
{
... lines 7 - 12
    private $_links = array();
... lines 14 - 25
}
```

Creating and Setting the Links

To actually add links, create a new function called public function addLink() that has two arguments: the $ref - that's the *name* of the link, like first or last - and the $url. Add the link with $this->_links[$ref] = $url;. Great - now head back to the controller:

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

```
... lines 1 - 4
class PaginatedCollection
{
... lines 7 - 21
    public function addLink($ref, $url)
    {
        $this->_links[$ref] = $url;
    }
}
```

Every link will point to the same route, but with a different page query parameter. The route to this controller doesn't have a name yet, so give it one: api_programmers_collection. Copy that name and set it to a $route variable:

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

```
... lines 1 - 21
class ProgrammerController extends BaseController
{
... lines 24 - 75
    /**
     * @Route("/api/programmers", name="api_programmers_collection")
     * @Method("GET")
     */
    public function listAction(Request $request)
    {
... lines 82 - 98
        $route = 'api_programmers_collection';
... lines 100 - 120
    }
... lines 122 - 220
}
```

Next, create $routeParams: this will hold any wildcards that need to be passed to the route - meaning the curly brace parts in

its path. This route doesn't have any, so set leave it empty. We're already setting things up to be reusable for other paginated responses.

Since we need to generate *four* links, create an anonymous function to help out with this: $createLinkUrl = function(). Give it one argument $targetPage. Also, add use for $route and $routeParams so we can access those inside. To generate the URL, use the normal return $this->generateURL() passing it the $route and an array_merge() of any routeParams with a new page key:

```
222 lines | src/AppBundle/Controller/Api/ProgrammerController.php
... lines 1 - 98
        $route = 'api_programmers_collection';
        $routeParams = array();
        $createLinkUrl = function($targetPage) use ($route, $routeParams) {
            return $this->generateUrl($route, array_merge(
                $routeParams,
                array('page' => $targetPage)
            ));
        };
... lines 107 - 222
```

Since there's no {page} routing wildcard, the router will add a ?page= query parameter to the end, exactly how we want it to.

Sweet! Add the first link with $paginatedCollection->addLink(). Call this link self and use $page to point to the *current* page. It might seem silly to link to *this* page, but it's a pretty standard thing to do:

```
222 lines | src/AppBundle/Controller/Api/ProgrammerController.php
... lines 1 - 107
        $paginatedCollection->addLink('self', $createLinkUrl($page));
... lines 109 - 222
```

Copy this line and paste it twice. Name the second link first instead of self and point this to page 1. Name the third link last and have it generate a URL to the last page: $pagerfanta->getNbPages():

```
222 lines | src/AppBundle/Controller/Api/ProgrammerController.php
... lines 1 - 108
        $paginatedCollection->addLink('first', $createLinkUrl(1));
        $paginatedCollection->addLink('last', $createLinkUrl($pagerfanta->getNbPages()));
... lines 111 - 222
```

The last two links are next and previous... but wait! We don't *always* have a next or previous page: these should be conditional. Add: if($pagerfanta->hasNextPage()), well, then, of course we want to generate a link to $pagerfanta->getNextPage() that's called next:

```
222 lines | src/AppBundle/Controller/Api/ProgrammerController.php
... lines 1 - 110
        if ($pagerfanta->hasNextPage()) {
            $paginatedCollection->addLink('next', $createLinkUrl($pagerfanta->getNextPage()));
        }
... lines 114 - 222
```

Do this same thing for the previous page. if($pagerfanta->hasPreviousPage()), then getPreviousPage() and call that link prev:

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

```
... lines 1 - 113
        if ($pagerfanta->hasPreviousPage()) {
            $paginatedCollection->addLink('prev', $createLinkUrl($pagerfanta->getPreviousPage()));
        }
... lines 117 - 222
```

Phew!

With some luck, the test should pass:

```
$ ./bin/phpunit -c app --filter testGETProgrammersCollectionPaginated
```

Rerun it aaaannnddd perfect! This is pretty cool: the tests actually *follow* those links: walking from page 1 to page 2 to page 3 and asserting things along the way.

Link rels (self, first, etc)

The link keys - self, first, last, next and prev are actually called link rels, or relations. They have a very important purpose: to explain the *meaning* of the link. On the web, the link's text tells us what that link points to. In an API, the "rel" does that job.

In other words, as long as our API client understands first means the first page of results and next means the next page of results, you can communicate the significance of what those links are.

And you know what else? I *didn't* just invent these link rels. They're super-official IANA rels - an organization that tries to standardize some of this stuff. Why is that cool? Because if everyone used these same links for pagination, understanding API's would be easier and more consistent.

We *are* going to talk about links a lot more in a future episode - including all those buzzwords like hypermedia and HATEOAS. So sit tight.

Chapter 4: Reusable Pagination System

Since pagination always looks the same, no matter what you're listing, I *really* want to organize my code so that pagination is *effortless* in the future. This took *way* too many lines of code.

Inside of the Pagination/ directory, create a new PHP class called PaginationFactory. There, add a new public function createCollection() method: this will create the *entire* final PaginatedCollection object for some collection resource. To do this, we'll need to pass it a few things, starting with the $qb and the $request - we'll use that to find the *current* page. The method will also need to know the route for the links and any $routeParams it needs:

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

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

use Doctrine\ORM\QueryBuilder;
... lines 6 - 7
use Symfony\Component\HttpFoundation\Request;
... lines 9 - 10
class PaginationFactory
{
... lines 13 - 19
    public function createCollection(QueryBuilder $qb, Request $request, $route, array $routeParams = array())
    {
... lines 22 - 53
    }
}
```

Go back to ProgrammerController, copy the logic, remove it and put it into PaginationFactory. Add the missing use statements: by auto-completing the classes DoctrineORMAdapter and Pagerfanta. Now, delete $route and $routeParams since those are passed as arguments. Remove the $qb variable for the same reason:

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

```
... lines 1 - 5
use Pagerfanta\Adapter\DoctrineORMAdapter;
use Pagerfanta\Pagerfanta;
... lines 8 - 10
class PaginationFactory
{
... lines 13 - 19
    public function createCollection(QueryBuilder $qb, Request $request, $route, array $routeParams = array())
    {
        $page = $request->query->get('page', 1);

        $adapter = new DoctrineORMAdapter($qb);
        $pagerfanta = new Pagerfanta($adapter);
        $pagerfanta->setMaxPerPage(10);
        $pagerfanta->setCurrentPage($page);

        $programmers = [];
        foreach ($pagerfanta->getCurrentPageResults() as $result) {
            $programmers[] = $result;
        }

        $paginatedCollection = new PaginatedCollection($programmers, $pagerfanta->getNbResults());

        $createLinkUrl = function($targetPage) use ($route, $routeParams) {
            return $this->router->generate($route, array_merge(
                $routeParams,
                array('page' => $targetPage)
            ));
        };

        $paginatedCollection->addLink('self', $createLinkUrl($page));
        $paginatedCollection->addLink('first', $createLinkUrl(1));
        $paginatedCollection->addLink('last', $createLinkUrl($pagerfanta->getNbPages()));
        if ($pagerfanta->hasNextPage()) {
            $paginatedCollection->addLink('next', $createLinkUrl($pagerfanta->getNextPage()));
        }
        if ($pagerfanta->hasPreviousPage()) {
            $paginatedCollection->addLink('prev', $createLinkUrl($pagerfanta->getPreviousPage()));
        }

        return $paginatedCollection;
    }
}
```

In fact, move that back to ProgrammerController: we'll want it in a minute:

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 76
    public function listAction(Request $request)
    {
        $qb = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findAllQueryBuilder();
... lines 82 - 84
        $response = $this->createApiResponse($paginatedCollection, 200);

        return $response;
    }
... lines 89 - 187
}
```

The only other problem here is $this->generateUrl(): that method does *not* exist outside of the controller. That's ok: since we *do* need to generate URLs, this just means we need the router. Add a __construct() function at the top with RouterInterface as an argument. I'll use the Alt + enter PHPStorm shortcut to create and set that property:

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

```
... lines 1 - 8
use Symfony\Component\Routing\RouterInterface;

class PaginationFactory
{
    private $router;

    public function __construct(RouterInterface $router)
    {
        $this->router = $router;
    }
... lines 19 - 54
}
```

Back inside createCollection(), change $this->generateUrl() to $this->router->generate():

55 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 - 35
        $createLinkUrl = function($targetPage) use ($route, $routeParams) {
            return $this->router->generate($route, array_merge(
                $routeParams,
                array('page' => $targetPage)
            ));
        };
... lines 42 - 53
    }
}
```

Our work in this class is done! Next, register this as service in app/config/services.yml - let's call it pagination_factory. How creative! Set the class to PaginationFactory and pass one key for arguments: @router:

29 lines | app/config/services.yml

```
... lines 1 - 5
6  services:
... lines 7 - 25
26     pagination_factory:
27         class: AppBundle\Pagination\PaginationFactory
28         arguments: ['@router']
```

Tip

If you're using Symfony 3.3, your app/config/services.yml contains some extra code that may break things when following this tutorial! To keep things working - and learn about what this code does - see https://knpuniversity.com/symfony-3.3-changes

Copy the service name and open ProgrammerController to hook this all up. Now, just use $paginatedCollection = $this->get('pagination_factory')->createCollection() and pass it the 4 arguments: $qb, $request, the route name - api_programmers_collection - and the route params:

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

```
... lines 1 - 18
19  class ProgrammerController extends BaseController
20  {
... lines 21 - 76
77      public function listAction(Request $request)
78      {
... lines 79 - 81
82          $paginatedCollection = $this->get('pagination_factory')
83              ->createCollection($qb, $request, 'api_programmers_collection');
... lines 84 - 87
88      }
... lines 89 - 187
188 }
```

Actually, most of the time you won't have route params. So head back into PaginationFactory and make that argument optional:

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

```
... lines 1 - 10
11  class PaginationFactory
12  {
... lines 13 - 19
20      public function createCollection(QueryBuilder $qb, Request $request, $route, array $routeParams = array())
21      {
... lines 22 - 53
54      }
55  }
```

Much better.

Now, PhpStorm *should* be happy... but it's still not! It looks more like someone stole it's ice cream. Ah, I forgot to return $paginatedCollection in PaginationFactory:

55 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 - 52
        return $paginatedCollection;
    }
}
```

PhpStorm was complaining that createCollection() didn't look like it returned anything... and it was right! The robots are definitely taking over.

Run the test to see if we broke anything:

```
$ ./bin/phpunit -c app --filter filterGETProgrammersCollectionPaginated
```

We didn't! What a delightful surprise.

Now, if you want some sweet pagination, just create a QueryBuilder, pass it into the PaginationFactory, pass that to createApiResponse and then go find some ice cream.

Chapter 5: Filtering / Searching

Designing how to Filter

Paginated a big collection is a must. But you might also want a client to be able to search or filter that collection. Ok, so how do we search on the web? Well usually, you fill in a box, hit submit, and that makes a GET request with your search term as a query parameter like ?q=. The server reads that and returns the results.

I have an idea! Let's do the *exact* same thing! First, we will *of course* add a test. Add a new programmer at the top of the pagination test with $this->createProgrammer(). I want to do a search that will *not* return this new programmer, but still *will* return the original 25. To do that, give it a totally different nickname, like 'nickname' => 'willnotmatch'. Keep the avatar number as 3... because we don't really care:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 71
    public function testGETProgrammersCollectionPaginated()
    {
        $this->createProgrammer(array(
            'nickname' => 'willnotmatch',
            'avatarNumber' => 5,
        ));
... lines 78 - 120
    }
... lines 122 - 226
}
```

For the query parameter, use whatever name you want: how about ?filter=programmer:

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

```
... lines 1 - 85
        // page 1
        $response = $this->client->get('/api/programmers?filter=programmer');
... lines 88 - 228
```

If you're feeling fancy, you could have multiple query parameters for different fields, or some cool search syntax like on GitHub. That's all up to you - the API will still work exactly the same.

Filtering the Collection

Great news: it turns out that this is going to be pretty easy. First, get the filter value: $filter = $request->query->get('filter');. Pass that to the "query builder" function as an argument. Let's update that to handle a filter string:

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 76
    public function listAction(Request $request)
    {
        $filter = $request->query->get('filter');

        $qb = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findAllQueryBuilder($filter);
... lines 84 - 89
    }
... lines 91 - 189
}
```

In ProgrammerRepository, add a $filter argument, but make it optional:

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

```
... lines 1 - 8
class ProgrammerRepository extends EntityRepository
{
... lines 11 - 28
    public function findAllQueryBuilder($filter = '')
    {
... lines 31 - 38
    }
}
```

Below, set the old return value to a new $qb variable. Then, if ($filter) has some value, add a where clause: andWhere('programmer.nickname LIKE :filter OR programmer.tagLine LIKE filter'). Then use setParameter('filter' , '%'.$filter.'%'). Finish things by returning $qb at the bottom:

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

```
... lines 1 - 8
class ProgrammerRepository extends EntityRepository
{
... lines 11 - 28
    public function findAllQueryBuilder($filter = '')
    {
        $qb = $this->createQueryBuilder('programmer');

        if ($filter) {
            $qb->andWhere('programmer.nickname LIKE :filter OR programmer.tagLine LIKE :filter')
                ->setParameter('filter', '%'.$filter.'%');
        }

        return $qb;
    }
}
```

If you were using something like Elastic Search, then you wouldn't be making this query through Doctrine: you'd be doing it through elastic search itself. But the idea is the same: prepare some search for Elastic, then use an Elastic Search adapter with Pagerfanta.

And that's all there is to it! Re-run the test:

```
$ ./bin/phpunit -c app --filter filterGETProgrammersCollectionPaginated
```

Oooh a 500 error: let's see what we're doing wrong:

> Parse error, unexpected '.' on ProgrammerRepository line 38.

Ah yes, it looks like I tripped over my keyboard. Delete that extra period and run this again:

```
$ ./bin/phpunit -c app --filter filterGETProgrammersCollectionPaginated
```

Hmm, it's *still* failing: this time when it goes to page 2. To debug, let's see what happens if we comment out the filter logic and try again:

```
$ ./bin/phpunit -c app --filter filterGETProgrammersCollectionPaginated
```

Now it fails on page 1: that extra willnotmatch programmer is returned and that makes index 5 Programmer4 instead of Programmer5. When we put the filter logic back, it has that exact same problem on page 2. Can you guess what's going on here? Yeas! We're losing our filter query parameter when we paginate through the results. womp womp.

Don't Lose the Filter Parameter!

In the test, the URL ends in ?page=2 with *no* filter on it. We need to maintain the filter query parameter *through* our pagination. Since we have everything centralized in, PaginationFactory that's going to be easy. Add $routeParams = array_merge() and merge $routeParams with all of the current query parameters, which is $request->query->all(). That should take care of it:

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());

        // make sure query parameters are included in pagination links
        $routeParams = array_merge($routeParams, $request->query->all());
... lines 38 - 56
    }
}
```

Run the tests one last time:

```
$ ./bin/phpunit -c app --filter filterGETProgrammersCollectionPaginated
```

And we're green for filtering!

Chapter 6: Serialization Event Subscriber

I think the best part of doing API magic in Symfony is the serializer we've been using. We just give it objects - whether those are entities or something else - and it takes care of turning its properties into JSON. And we have control too: by using the exclusion policy and other annotations like @SerializedName that lets us control the JSON key a property becomes.

When does the Serializer Fail?

Heck, we can even add virtual properties! Just add a function inside your class, add the @VirtualProperty() annotation above it... and bam! You now have another field in your JSON response that's not *actually* a property on the class. That's great! It handles 100% of what we need! Right... right?

Ah, ok: there's still this last, nasty 1% of use-cases where virtual property won't work. Why? Well, imagine you want to include the URL to the programmer in its JSON representation. To make that URL, you need the router service. But can you access services from within a method in Programmer? No! We're in trouble!

This is usually where I get really mad and say "Never mind, I'm not using the stupid serializer anymore!" Then I stomp off to my bedroom to play video games.

But come on, we can definitely overcome this. In fact, there are *two* ways. The more interesting is with an event subscriber on the serializer.

Creating a Serializer Event Subscriber

In AppBundle, create a new directory called Serializer and put a fancy new class inside called LinkSerializationSubscriber. Set the namespace to AppBundle\Serializer:

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

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

use JMS\Serializer\EventDispatcher\EventSubscriberInterface;

class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 9 - 11
}
```

To create a subscriber with the JMSSerializer, you need to implement EventSubscriberInterface... and make sure you choose the one from JMS\Serializer. There's also a core interface that, unfortunately, has the exact same name.

In PhpStorm, I'll open the Generate shortcut and select "Implement Methods". This will tell me all the methods that the interface requires. And, it's just one: getSubscribedEvents:

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

```
... lines 1 - 6
class LinkSerializationSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
    }
}
```

Stop: here's the goal. Whenever we serialize something, there are a few events we can hook into to customize that process. In this method, we'll tell the serializer exactly which events we want to hook into. One of those will allow us to *add* a new field... which will be the URL to whatever Programmer is being serialized.

Return an array with another array inside: we'll need a few keys here. The first is event - the event name we need to hook into. There are two for serialization: serializer.pre_serialize and serializer.post_serialize:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 8
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 11 - 17
    public static function getSubscribedEvents()
    {
        return array(
            array(
                'event' => 'serializer.post_serialize',
... lines 23 - 25
            )
        );
    }
}
```

We need the second because it lets you *change* the data that's being turned into JSON.

Add a method key and set it to onPostSerialize - we'll create that in a second:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 21
                'event' => 'serializer.post_serialize',
                'method' => 'onPostSerialize',
... lines 24 - 30
```

Next, add format set to JSON:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 21
                'event' => 'serializer.post_serialize',
                'method' => 'onPostSerialize',
                'format' => 'json',
... lines 25 - 30
```

This means the method will *only* be called when we're serializing into JSON... which is fine - that's all our API does.

Finally, add a class key set to AppBundle\Entity\Programmer:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 21
                'event' => 'serializer.post_serialize',
                'method' => 'onPostSerialize',
                'format' => 'json',
                'class' => 'AppBundle\Entity\Programmer'
... lines 26 - 30
```

This says, "Hey! Only call onPostSerialize for Programmer classes!".

Adding a Custom Serialized Field

Setup, done! Create that public function onPostSerialize(). Just like with core Symfony events, you'll be passed an event argument, which in this case is an instance of ObjectEvent:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 8
class LinkSerializationSubscriber implements EventSubscriberInterface
{
    public function onPostSerialize(ObjectEvent $event)
    {
... lines 13 - 15
    }
... lines 17 - 28
}
```

Now, we can start messing with the serialization process.

Before we go any further, go back to our test. The goal is for each Programmer to have a new field that is a link to itself. In testGETProgrammer(), add a new assert that checks that we have a uri property that's equal to /api/programmers/UnitTester:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 35
    public function testGETProgrammer()
    {
... lines 38 - 51
        $this->asserter()->assertResponsePropertyEquals($response, 'uri', '/api/programmers/UnitTester');
    }
... lines 54 - 227
}
```

Ok, let's see how we can use the fancy subscriber to add this field automatically.

Chapter 7: Super Custom Serialization Fields

The Serialization Visitor

Back in the subscriber, create a new variable called $visitor and set it to $event->getVisitor(). The visitor is kind of in charge of the serialization process. And since we *know* we're serializing to JSON, this will be an instance of JsonSerializationVisitor. Write an inline doc for that and add a use statement up top. That will give us autocompletion:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 6
use JMS\Serializer\JsonSerializationVisitor;

class LinkSerializationSubscriber implements EventSubscriberInterface
{
    public function onPostSerialize(ObjectEvent $event)
    {
        /** @var JsonSerializationVisitor $visitor */
        $visitor = $event->getVisitor();
... line 15
    }
... lines 17 - 28
}
```

Oh, hey, look at this - that class has a method on it called addData(). We can use it to add whatever cool custom fields we want. Add that new uri field, but just set it to the classic FOO value for now:

30 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
        /** @var JsonSerializationVisitor $visitor */
        $visitor = $event->getVisitor();
        $visitor->addData('uri', 'FOO');
... lines 16 - 30
```

Registering the Subscriber

The *last* thing we need to do - which you can probably guess - is register this as a service. In services.yml, add the service - how about link_serialization_subscriber. Add the class and skip arguments - we don't have any yet. But we *do* need a tag so that the JMS Serializer knows about our class. Set the tag name to jms_serializer.event_subscriber:

34 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 29
    link_serialization_subscriber:
        class: AppBundle\Serializer\LinkSerializationSubscriber
        tags:
            - { name: jms_serializer.event_subscriber }
```

Ok, try the test! Copy the method name, head to the terminal and run:

```
$ ./bin/phpunit -c app --filter testGETProgrammer
```

and then paste in the name. This method name matches a few tests, so we'll see more than just our *one* test run. Yes, it

fails... but in a good way!

```
FOO does not match /api/programmers/UnitTester.
```

Above, we *do* have the new, custom uri field.

Making the URI Dynamic

This means we're *almost* done. To generate the real URI, we need the router. Add the __construct() method with a RouterInterface argument. I'll use the option+enter shortcut to create that property and set it:

47 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 7
use Symfony\Component\Routing\RouterInterface;
... lines 9 - 10
class LinkSerializationSubscriber implements EventSubscriberInterface
{
    private $router;

    public function __construct(RouterInterface $router)
    {
        $this->router = $router;
    }
... lines 19 - 45
}
```

In onPostSerialize() say $programmer = $event->getObject();. Because of our configuration below, we *know* this will only be called when the object is a Programmer. Add some inline documentation for the programmer and plug in its use statement:

47 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 8
use AppBundle\Entity\Programmer;

class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onPostSerialize(ObjectEvent $event)
    {
        /** @var JsonSerializationVisitor $visitor */
        $visitor = $event->getVisitor();
        /** @var Programmer $programmer */
        $programmer = $event->getObject();
... lines 26 - 32
    }
... lines 34 - 45
}
```

Finally, for the data type $this->router->generate() and pass it api_programmers_show and an array containing nickname set to $programmer->getNickname():

47 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 10
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onPostSerialize(ObjectEvent $event)
    {
... lines 22 - 26
        $visitor->addData(
            'uri',
            $this->router->generate('api_programmers_show', [
                'nickname' => $programmer->getNickname()
            ])
        );
    }
... lines 34 - 45
}
```

Cool! Now, go back to services.yml and add an arguments key with just @router:

35 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 29
    link_serialization_subscriber:
        class: AppBundle\Serializer\LinkSerializationSubscriber
        arguments: ['@router']
        tags:
            - { name: jms_serializer.event_subscriber }
```

Ok, moment of truth! Run the test!

```
$ ./bin/phpunit -c app --filter testGETProgrammer
```

And... it's failing. Ah, the URL has ?nickname=UnitTester. Woh woh. I bet that's my fault. The name of the route in onPostSerialize() should be api_programmers_show:

47 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 10
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onPostSerialize(ObjectEvent $event)
    {
... lines 22 - 26
        $visitor->addData(
            'uri',
            $this->router->generate('api_programmers_show', [
                'nickname' => $programmer->getNickname()
            ])
        );
    }
... lines 34 - 45
}
```

Re-run the test:

```
$ ./bin/phpunit -c app --filter testGETProgrammer
```

It's still failing, but for a new reason. This time it doesn't like the app_test.php at the beginning of the link URI. Where's that coming from?

The test class extends an ApiTestCase: we made this in an earlier episode. This app already has a test environment and it configures a *test* database connection. If we can force every URL through app_test.php, it'll use that test database, and we'll be really happy:

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

```
... lines 1 - 20
class ApiTestCase extends KernelTestCase
{
... lines 23 - 46
    public static function setUpBeforeClass()
    {
... lines 49 - 59
        // guaranteeing that /app_test.php is prefixed to all URLs
        self::$staticClient->getEmitter()
            ->on('before', function(BeforeEvent $event) {
                $path = $event->getRequest()->getPath();
                if (strpos($path, '/api') === 0) {
                    $event->getRequest()->setPath('/app_test.php'.$path);
                }
            });
... lines 68 - 69
    }
... lines 71 - 295
}
```

We did a cool thing with Guzzle to accomplish this: automatically prefixing our requests with app_test.php. But because of that, when we generate URLs, they will also have app_test.php. That's a good thing in general, just not when we're comparing URLs in a test.

Copy that path and create a helper function at the bottom of ApiTestCase called protected function adjustUri(). Make this return /app_test.php plus the $uri. This method can help when comparing expected URI's:

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

```
... lines 1 - 20
class ApiTestCase extends KernelTestCase
{
... lines 23 - 282
    /**
     * 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;
    }

}
```

Now, in ProgrammerControllerTest, just wrap the expected URI in $this->adjustUri():

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 35
    public function testGETProgrammer()
    {
... lines 38 - 51
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            'uri',
            $this->adjustUri('/api/programmers/UnitTester')
        );
    }
... lines 58 - 231
}
```

This isn't a particularly incredible solution, but now we can properly test things. Run the tests again...

```
$ ./bin/phpunit -c app --filter testGETProgrammer
```

And... It's green! Awesome!

Method 2: Adding Custom Fields

One last thing! I mentioned that there are *two* ways to add super-custom fields like uri. Using a serializer subscriber is the first. But sometimes, your API representation will look *much* different than your entity. Imagine we had some crazy endpoint that returned info about a Programmer mixed with details about their last 3 battles, the last time they fought and the current weather in their hometown.

Can you imagine trying to do this? You'll need multiple @VirtualProperty methods and probably some craziness inside an event subscriber. It might work, but it'll look ugly and be confusing.

In this case, there's a much better way: create a new class with the exact properties you need. Then, instantiate it, populate the object in your controller and serialize it. This class isn't an entity - it's just there to model your API response. I *love* this approach and recommend it as soon as you're doing more than just a few serialization customizations to a class.

Chapter 8: Adding Links via Annotations

Oh man, this chapter will be one of my *favorite* ever to record, because we're going to do some sweet stuff with annotations.

In ProgrammerControllerTest, we called this key uri:

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

```
... lines 1 - 5
6   class ProgrammerControllerTest extends ApiTestCase
7   {
... lines 8 - 35
36      public function testGETProgrammer()
37      {
... lines 38 - 51
52          $this->asserter()->assertResponsePropertyEquals(
53              $response,
54              'uri',
55              $this->adjustUri('/api/programmers/UnitTester')
56          );
57      }
... lines 58 - 231
232 }
```

Because, well... why not?

But when we added pagination, we included *its* links inside a property called _links:

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

```
... lines 1 - 5
6   class ProgrammerControllerTest extends ApiTestCase
7   {
... lines 8 - 76
77      public function testGETProgrammersCollectionPaginated()
78      {
... lines 79 - 101
102         $this->asserter()->assertResponsePropertyExists($response, '_links.next');
... lines 103 - 125
126     }
... lines 127 - 231
232 }
```

That kept links separate from data. I think we should do the same thing with uri: change it to _links.self. The key self is a name used when linking to, your, "self":

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 35
    public function testGETProgrammer()
    {
... lines 38 - 51
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.self',
            $this->adjustUri('/api/programmers/UnitTester')
        );
    }
... lines 58 - 231
}
```

Renaming this is easy, but we have a bigger problem. Adding links is too much work. Most importantly, the subscriber only works for Programmer objects - so we'll need more event listeners in the future for other classes.

I have a different idea. Imagine we could link via annotations, like this: add @Link with "self" inside, route = "api_programmers_show" params = { }. This route has a nickname wildcard, so add "nickname": and then use the expression object.getNickname():

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

```
... lines 1 - 9
/**
 * Programmer
... lines 12 - 15
 * @Link(
 *  "self",
 *  route = "api_programmers_show",
 *  params = { "nickname": "object.getNickname()" }
 * )
 */
class Programmer
{
... lines 24 - 194
}
```

That last part is an *expression*, from Symfony's expression language. You and I are going to build the system that makes this work, so I'm going to assume that we'll pass a variable called object to the expression language that is this Programmer object being serialized. Then, we just call .getNickname().

Of course, this won't work yet - in fact it'll totally bomb if you try it. But it will in a few minutes!

Creating an Annotation

To create this cool system, we need to understand a bit about annotations. Every annotation - like Table or Entity from Doctrine - has a class behind it. That means *we* need a Link class. Create a new directory called Annotation. Inside add a new Link class in the AppBundle\Annotation namespace:

27 lines | src/AppBundle/Annotation/Link.php

```
<?php

namespace AppBundle\Annotation;
... lines 4 - 8
class Link
{
... lines 11 - 25
}
```

To hook this annotation into the annotations system, we need a few annotations: the first being, um, well, @Annotation. Yep, I'm being serious. The second is @Target, which will be "CLASS". This means that this annotation is expected to live above class declarations:

27 lines | src/AppBundle/Annotation/Link.php

```
... lines 1 - 4
/**
 * @Annotation
 * @Target("CLASS")
 */
class Link
{
... lines 11 - 25
}
```

Inside the Link class, we need to add a public property for *each* option that can be passed to the annotation, like route and params. Add public $name;, public $route; and public $params = array();:

27 lines | src/AppBundle/Annotation/Link.php

```
... lines 1 - 8
class Link
{
... lines 11 - 15
    public $name;
... lines 17 - 22
    public $route;

    public $params = array();
}
```

The first property becomes the default property, which is why we don't need to have name = "self" when using it.

The name and route options are required, so add an extra @Required above them:

27 lines | src/AppBundle/Annotation/Link.php

```
... lines 1 - 8
class Link
{
    /**
     * @Required
     *
     * @var string
     */
    public $name;

    /**
     * @Required
     *
     * @var string
     */
    public $route;
... lines 24 - 25
}
```

And... that's it!

Inside of Programmer, every annotation - except for the special @Annotation and @Target guys, they're core to that system - needs a use statement - we already have some for @Serializer, @Assert and @ORM. Add a use statement directly to the class itself for @Link:

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

```
... lines 1 - 7
use AppBundle\Annotation\Link;

/**
 * Programmer
 *
 * @ORM\Table(name="battle_programmer")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\ProgrammerRepository")
 * @Serializer\ExclusionPolicy("all")
 * @Link(
 *  "self",
 *  route = "api_programmers_show",
 *  params = { "nickname": "object.getNickname()" }
 * )
 */
class Programmer
{
... lines 24 - 194
}
```

This hooks the annotation up with the class we just created.

Reading the Annotation

Ok... so how do we read annotations? Great question, I have no idea. Ah, it's easy, thanks to the Doctrine annotations library that comes standard with Symfony. In fact, we already have a service available called @annotation_reader.

Inside LinkSerializationSubscriber, inject that as the second argument. It's an instance of the Reader interface from Doctrine\Common\Annotations. Call it $annotationsReader:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 5
use Doctrine\Common\Annotations\Reader;
... lines 7 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
    private $router;

    private $annotationReader;
... lines 18 - 20
    public function __construct(RouterInterface $router, Reader $annotationReader)
    {
        $this->router = $router;
        $this->annotationReader = $annotationReader;
... line 25
    }
... lines 27 - 72
}
```

I'll hit option+enter and select initialize fields to get that set on property.

And before I forget, in services.yml, inject that by adding @annotation_reader as the second argument:

35 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 29
    link_serialization_subscriber:
        class: AppBundle\Serializer\LinkSerializationSubscriber
        arguments: ['@router', '@annotation_reader']
        tags:
            - { name: jms_serializer.event_subscriber }
```

Super easy.

Too easy, back to work! Delete all of this junk in onPostSerialize() and start with $object = $event->getObject(). To read the annotations off of that object, add $annotations = $this->annotationReader->getClassAnnotations(). Pass that a new \ReflectionObject() for $object:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 27
    public function onPostSerialize(ObjectEvent $event)
    {
        /** @var JsonSerializationVisitor $visitor */
        $visitor = $event->getVisitor();

        $object = $event->getObject();
        $annotations = $this->annotationReader
            ->getClassAnnotations(new \ReflectionObject($object));
... lines 36 - 50
    }
... lines 52 - 72
}
```

That's it!

Now, the class *could* have a lot of annotations above it, but we're only interested in the @Link annotation. We'll add an if statement to look for that in a second. But first, create $links = array(): that'll be our holder for any links we find:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 32
        $object = $event->getObject();
        $annotations = $this->annotationReader
            ->getClassAnnotations(new \ReflectionObject($object));

        $links = array();
... lines 38 - 74
```

Now, foreach ($annotations as $annotations). Immediately, see if this is something we care about with if ($annotation instanceof Link). At this point, the annotation options are populated on the public properties of the Link object. To get the URI, we can just say $this->router->generate() and pass it $annotation->route and $annotation->params:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 36
        $links = array();
        foreach ($annotations as $annotation) {
            if ($annotation instanceof Link) {
                $uri = $this->router->generate(
                    $annotation->route,
                    $this->resolveParams($annotation->params, $object)
                );
... line 44
            }
        }
... lines 47 - 74
```

How sweet is that? Well, we're not done yet: the params contain an expression string... which we're not parsing yet. We'll get back to that in a second.

Finish this off with $links[$annotation->name] = $uri;. At the bottom, finish with the familiar $visitor->addData() with _links set to $links;. Other than evaluating the expression, that's all the code you need:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 27
    public function onPostSerialize(ObjectEvent $event)
    {
... lines 30 - 36
        $links = array();
        foreach ($annotations as $annotation) {
            if ($annotation instanceof Link) {
                $uri = $this->router->generate(
                    $annotation->route,
                    $this->resolveParams($annotation->params, $object)
                );
                $links[$annotation->name] = $uri;
            }
        }

... line 48
            $visitor->addData('_links', $links);
... line 50
    }
... lines 52 - 72
}
```

Check this out by going to /api/programmers in the browser. Look at that! The embedded programmer entities actually have a link called self. It worked!

Of course, the link is totally wrong because we're not evaluating the expression yet. But, we're really close.

Chapter 9: Evaluating the Link Expression

Before we fix the expression stuff, remove the class option from getSubscribedEvents() because we want this to be called for *all* classes:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 62
    public static function getSubscribedEvents()
    {
        return array(
            array(
                'event' => 'serializer.post_serialize',
                'method' => 'onPostSerialize',
                'format' => 'json',
            )
        );
    }
}
```

Avoiding Duplicated _links

When you do that, things still work. But now: clear your cache in the terminal:

```
$ ./app/console cache:clear
```

That's not normally something you need to do - but the JMSSerializerBundle doesn't properly update its cache when you change this option. Refresh again.

Ah, huge error! Apparently there is already data for _links!? That's a bit weird.

Ah, but wait: one of the things we're serializing is the paginated collection itself, which already has a _links property.

For better or worse, the JMS serializer library doesn't let you overwrite data on a field. To fix this, add an if statement that only adds _links if we found some on the object:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 27
    public function onPostSerialize(ObjectEvent $event)
    {
... lines 30 - 47
        if ($links) {
            $visitor->addData('_links', $links);
        }
    }
... lines 52 - 72
}
```

There's an even better fix - but I'll leave it to you. That would be to go into PaginatedCollection and replace its links with the

@Link annotation. This would be a little bit of work, but I believe in you!

Evaluating the Expression

Refresh the browser again. Things look good! Time to evaluate the expression.

To use the expression language, we just need to create an ExpressionLanguage object. We *could* register this as a service, but I'll take a shortcut and instantiate a new expression language right inside the constructor: $this->expressionEngine = new ExpressionLanguage():

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 18
    private $expressionLanguage;
... line 20
    public function __construct(RouterInterface $router, Reader $annotationReader)
    {
... lines 23 - 24
        $this->expressionLanguage = new ExpressionLanguage();
    }
... lines 27 - 72
}
```

That class lives in the ExpressionLanguage component.

Rename that property to expressionLanguage. Later, if I *do* want to register this as a service instead of creating it new right here, that'll be really easy.

Wrap $annotation->params in a call to $this->resolveParams() and pass it the params *and* the object, since we'll need to pass that into the expression itself:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 27
    public function onPostSerialize(ObjectEvent $event)
    {
... lines 30 - 37
        foreach ($annotations as $annotation) {
            if ($annotation instanceof Link) {
                $uri = $this->router->generate(
                    $annotation->route,
                    $this->resolveParams($annotation->params, $object)
                );
                $links[$annotation->name] = $uri;
            }
        }
... lines 47 - 50
    }
... lines 52 - 72
}
```

Add the new private function resolveParams() and then loop over $params: foreach ($params as $key => $param):

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 52
    private function resolveParams(array $params, $object)
    {
        foreach ($params as $key => $param) {
... lines 56 - 57
        }
... lines 59 - 60
    }
... lines 62 - 72
}
```

For each param, we'll replace it with $this->expressionLanguage->evaluate(). Pass it $param - that's the expression. Next, since the expressions are expecting a variable called object, pass an array as the second argument with an object key set to $object. And let's not forget our $object argument to this method!

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 52
    private function resolveParams(array $params, $object)
    {
        foreach ($params as $key => $param) {
            $params[$key] = $this->expressionLanguage
                ->evaluate($param, array('object' => $object));
        }
... lines 59 - 60
    }
... lines 62 - 72
}
```

Finally, wrap this up return $params;. Now, each parameter is evaluated through the expression language, which is a lot like Twig:

74 lines | src/AppBundle/Serializer/LinkSerializationSubscriber.php

```
... lines 1 - 12
class LinkSerializationSubscriber implements EventSubscriberInterface
{
... lines 15 - 52
    private function resolveParams(array $params, $object)
    {
... lines 55 - 59
        return $params;
    }
... lines 62 - 72
}
```

Ok, back to the browser. There it is! How about our test?

```
$ ./bin/phpunit -c app --filter testGETProgrammer
```

Hey, they're passing too! Amazing!

In just a few short minutes, we made an entirely reusable linking system. I will admit that this idea was stolen from a library called Hateoas. Now, don't you feel dangerous?

Chapter 10: Conditionally Serializing Fields with Groups

Once upon a time, I worked with a client that had a really interesting API requirement. In fact, one that *totally* violate REST... but it's kinda cool. They said:

> When we have one object that relates to another object - like how our programmer relates to a user - *sometimes* we want to embed the user in the response and sometimes we don't. In fact, we want the API client to tell us via - a query parameter - whether or not they want embedded objects in the response.

Sounds cool...but it *totally* violates REST because you now have two different URLs that return the same resource... each just returns a different *representation*. Rules are great - but come on... if this is useful to you, make it happen.

Testing the Deep Functionality

Let's start with a quick test: copy part of testGETProgramer() and name the new method testGETProgrammerDeep(). Now, add a query parameter called ?deep:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 58
    public function testGETProgrammerDeep()
    {
        $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));

        $response = $this->client->get('/api/programmers/UnitTester?deep=1');
        $this->assertEquals(200, $response->getStatusCode());
... lines 68 - 70
    }
... lines 72 - 245
}
```

The idea is simple: if the client adds ?deep=1, then the API should expose more embedded objects. Use the asserter to say assertResponsePropertyExists(), pass that the $response and the property we'll expect, which is user. Since this will be an entire user object, check specifically for user.username:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 58
    public function testGETProgrammerDeep()
    {
... lines 61 - 67
        $this->asserter()->assertResponsePropertiesExist($response, array(
            'user.username'
        ));
    }
... lines 72 - 245
}
```

Very nice!

Serialization Groups

If you look at this response in the browser, we definitely do *not* have a user field. But there are only *two* little things we need to do to add it.

First, expose the user property with @Serializer\Expose():

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

```
... lines 1 - 21
22  class Programmer
23  {
... lines 24 - 65
66      /**
... lines 67 - 69
70       * @Serializer\Expose()
71       */
72      private $user;
... lines 73 - 196
197 }
```

Of course, it can't be that simple: now the user property would *always* be included. To avoid that, add @Serializer\Groups() and use a new group called deep:

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

```
... lines 1 - 21
22  class Programmer
23  {
... lines 24 - 65
66      /**
... lines 67 - 68
69       * @Serializer\Groups({"deep"})
70       * @Serializer\Expose()
71       */
72      private $user;
... lines 73 - 196
197 }
```

Here's the idea: when you serialize, each property belongs to one or more "groups". If you don't include the @Serializer\Groups annotation above a property, then it will live in a group called Default - with a capital D. Normally, the serializer serializes *all* properties, regardless of their group. But you can also tell it to serialize only the properties in a different group, or even in a set of groups. We can use groups to serialize the user property under only *certain* conditions.

But before we get there - I just noticed that the password field is being exposed on my User. That's definitely lame. Fix it by adding the Expose use statement, removing that last part and writing as Serializer instead. That's a nice trick to get that use statement:

86 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 7
8   use JMS\Serializer\Annotation as Serializer;
... lines 9 - 14
15  class User implements UserInterface
16  {
... lines 17 - 84
85  }
```

Now set @Serializer\ExclusionPolicy() above the class with all and add @Expose above username:

86 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 9
/**
 * @Serializer\ExclusionPolicy("all")
 * @ORM\Table(name="battle_user")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UserRepository")
 */
class User implements UserInterface
{
... lines 17 - 23
    /**
     * @Serializer\Expose()
     * @ORM\Column(type="string", unique=true)
     */
    private $username;
... lines 29 - 84
}
```

Back in Programmer.php, remove the "groups" code temporarily and refresh. OK good, *only* the username is showing. Put that "groups" code back.

Setting the SerializationGroup

Ok... so now, how can we serialize a specific set of groups? To answer that, open ProgrammerController and find showAction(). Follow createApiResponse() into the BaseController and find serialize():

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

```
... lines 1 - 16
abstract class BaseController extends Controller
{
... lines 19 - 123
    protected function serialize($data, $format = 'json')
    {
        $context = new SerializationContext();
        $context->setSerializeNull(true);

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

When we serialize, we create this SerializationContext, which holds a few options for serialization. Honestly, there's not much you can control with this, but you *can* set which *groups* you want to serialize.

First, get the $request object by fetching the request_stack service and adding getCurrentRequest(). Next, create a new $groups variable and set it to only Default: we *always* want to serialize the properties in this group:

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

```
... lines 1 - 16
17  abstract class BaseController extends Controller
18  {
... lines 19 - 123
124     protected function serialize($data, $format = 'json')
125     {
126         $context = new SerializationContext();
127         $context->setSerializeNull(true);
128
129         $request = $this->get('request_stack')->getCurrentRequest();
130         $groups = array('Default');
... lines 131 - 137
138     }
139 }
```

Now say if ($request->query->get('deep')) is true then add deep to $groups. Finish this up with $context->setGroups($groups):

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

```
... lines 1 - 16
17  abstract class BaseController extends Controller
18  {
... lines 19 - 123
124     protected function serialize($data, $format = 'json')
125     {
... lines 126 - 129
130         $groups = array('Default');
131         if ($request->query->get('deep')) {
132             $groups[] = 'deep';
133         }
134         $context->setGroups($groups);
... lines 135 - 137
138     }
139 }
```

Go Deeper!

You could also use $request->query->getBoolean('deep') instead of get() to convert the deep query parameter into a boolean. See accessing request data for other useful methods.

And just like that, we're able to conditionally show fields. Sweet!

Re-run our test for testGETProgrammerDeep():

```
$ ./bin/phpunit -c app --filter testGETProgrammer
```

It passes! To really prove it, refresh the browser. Nope, no user property. Now add ?deep=1 to the URL. That's a cool way to leverage groups.

Wow, nice work guys! We've just taken another huge chunk out of our API with pagination, filtering and a whole lot of cool serialization magic. Ok, now keep going with the next episode!