Symfony RESTful API: Errors (Course 2)

With <3 from SymfonyCasts

Chapter 1: Validation Errors 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!

Errors! A lot of things can go wrong - like 500 errors when your database server is on fire, 404 errors, validation errors, authentication errors and errors in judgement, like wearing a khaki shirt with khaki shorts, unless you're on Safari.

In your API, handling all of these errors correctly ends up taking some effort. So that's why we're devoting an entire episode on getting a beautiful, robust error system into your Symfony API.

First, we'll talk about what most people think of when you mention errors: validation errors! In episode 1, we created this cool ProgrammerControllerTest class where we can test all of our endpoints for creating a programmer, updating a programmer deleting a programmer, etc etc. We don't have validation on any of these endpoints yet.

Test for a Required Username

So let's add some: when we POST to create, we really need to make sure that the username isn't blank. That would be crazy! Copy testPOST(). Down at the bottom, paste that, and rename it to testValidationErrors(). Get rid of the nickname data field and most of the asserts:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 123
    public function testValidationErrors()
    {
        $data = array(
            'avatarNumber' => 2,
            'tagLine' => 'I\'m from a test!'
        );

        // 1) Create a programmer resource
        $response = $this->client->post('/api/programmers', [
            'body' => json_encode($data)
        ]);
... lines 135 - 144
    }
}
```

Validation Error Status Code: 400

Ok, design time! Writing the test is our time to *think* about how each endpoint should work. Since we're not sending a username, what status code should the endpoint return? Use 400:

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

```
... lines 1 - 123
    public function testValidationErrors()
    {
... lines 126 - 131
        $response = $this->client->post('/api/programmers', [
            'body' => json_encode($data)
        ]);

        $this->assertEquals(400, $response->getStatusCode());
... lines 137 - 144
    }
... lines 146 - 147
```

There are a few other status codes that you could use and we talk about them in our original REST series. But 400 is a solid choice.

Validation Errors Response Body

Next, what should the JSON content of our response hold? Let me suggest a format. Trust me for now - I'll explain why soon. We need to tell the client what went wrong - a validation error - and what the validation errors are.

Use $this->asserter()->assertResponsePropertiesExist() to assert that the response will have 3 properties. The first is type - a string error code - the second is title - a human description of what went wrong - and the third is errors - an array of all of the validation errors:

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

```
... lines 1 - 123
    public function testValidationErrors()
    {
... lines 126 - 131
        $response = $this->client->post('/api/programmers', [
            'body' => json_encode($data)
        ]);
... lines 135 - 136
        $this->asserter()->assertResponsePropertiesExist($response, array(
            'type',
            'title',
            'errors',
        ));
... lines 142 - 144
    }
... lines 146 - 147
```

Don't forget to pass the $response as the first argument. Now, think about that errors array property. I'm thinking it'll be an associative array where the keys are the fields that have errors, and the values are those errors. Use the asserter again to say assertResponsePropertyExists() to assert that errors.nickname exists:

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

```
... lines 1 - 123
    public function testValidationErrors()
    {
... lines 126 - 141
        $this->asserter()->assertResponsePropertyExists($response, 'errors.nickname');
... lines 143 - 144
    }
... lines 146 - 147
```

Basically, we want to assert that there *is* some error on the nickname field, because it should be required. Actually, go one step further and assert the exact validation message. Use assertResponsePropertyEquals with $response as the first argument, errors.nickname[0] as the second and, for the third, a nice message, how about "Please enter a clever nickname":

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

```
... lines 1 - 123
    public function testValidationErrors()
    {
... lines 126 - 141
        $this->asserter()->assertResponsePropertyExists($response, 'errors.nickname');
        $this->asserter()->assertResponsePropertyEquals($response, 'errors.nickname[0]', 'Please enter a clever nickname');
... line 144
    }
... lines 146 - 147
```

Why the [0] part? It won't be too common, but one field could have multiple errors, like a username that contains invalid characters *and* is too short. So each field will have an array of errors, and we're checking that the first is set to our clever message.

And since we *are* sending a valid avatarNumber, let's make sure that there is *no* error for it. Use assertResponsePropertyDoesNotExist() and pass it errors.avatarNumber:

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

```
... lines 1 - 123
    public function testValidationErrors()
    {
... lines 126 - 141
        $this->asserter()->assertResponsePropertyExists($response, 'errors.nickname');
        $this->asserter()->assertResponsePropertyEquals($response, 'errors.nickname[0]', 'Please enter a clever nickname');
        $this->asserter()->assertResponsePropertyDoesNotExist($response, 'errors.avatarNumber');
    }
... lines 146 - 147
```

Love it! We've just planned how we want our validation errors to work. That'll make coding this a lot easier.

Chapter 2: Sending back Validation Errors

Time to add validation errors and get this test passing. First, add the validation. Open the Programmer class. There's no validation stuff here yet, so we need the use statement for it. I'll use the NotBlank class directly, let PhpStorm auto-complete that for me, then remove the last part and add as Assert:

```
190 lines | src/AppBundle/Entity/Programmer.php
... lines 1 - 6
7  use Symfony\Component\Validator\Constraints as Assert;
... lines 8 - 190
```

That's a little shortcut to get that use statement you always need for validation.

Now, above username, add @Assert\NotBlank with a message option. Go back and copy the clever message and paste it here:

```
190 lines | src/AppBundle/Entity/Programmer.php
... lines 1 - 15
16  class Programmer
17  {
... lines 18 - 26
27      /**
... lines 28 - 31
32       * @Assert\NotBlank(message="Please enter a clever nickname")
33       */
34      private $nickname;
... lines 35 - 188
189 }
```

Handling Validation in the Controller

Ok! That was step 1. Step 2 is to go into the controller and send the validation errors back to the user. We're using forms, so handling validation is going to look pretty much identical to how it looks in traditional web forms.

Check out that processForm() function we created in episode 1:

```
145 lines | src/AppBundle/Controller/Api/ProgrammerController.php
... lines 1 - 15
16  class ProgrammerController extends BaseController
17  {
... lines 18 - 136
137     private function processForm(Request $request, FormInterface $form)
138     {
139         $data = json_decode($request->getContent(), true);
140
141         $clearMissing = $request->getMethod() != 'PATCH';
142         $form->submit($data, $clearMissing);
143     }
144 }
```

All this does is json_decode the request body and call $form->submit(). That does the same thing as $form->handleRequest(), which you're probably familiar with. So after this function is called, the form processing has happened. After it, add an if statement with the normal if (!$form->isValid()):

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

```
... lines 1 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 21
    public function newAction(Request $request)
    {
... lines 24 - 25
        $this->processForm($request, $form);

        if (!$form->isValid()) {
... lines 29 - 30
        }
... lines 32 - 46
    }
... lines 48 - 143
}
```

Tip

Calling just $form->isValid() before submitting the form is deprecated and will raise an exception in Symfony 4.0. Use $form->isSubmitted() && $form->isValid() instead to avoid the exception.

If we find ourselves here, it means we have validation errors. Let's see if this is working. Use the dump() function and the $form->getErrors() function, passing it true and false as arguments. That'll give us all the errors in a big tree. Cast this to a string - getErrors() returns a FormErrorIterator object, which has a nice __toString() method. Add a die at the end:

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

```
... lines 1 - 21
    public function newAction(Request $request)
    {
... lines 24 - 27
        if (!$form->isValid()) {
... line 29
            dump((string) $form->getErrors(true, false));die;
        }
... lines 32 - 46
    }
... lines 48 - 145
```

Let's run our test to see what this looks like. Copy the testValidationErrors method name, then run:

```
$ php bin/phpunit -c app --filter testValidationErrors
```

Ok, there's our printed dump. Woh, that is ugly. That's the nice HTML formatting that comes from the dump() function. But it's unreadable here. I'll show you a trick to clean that up.

It's dumping HTML because it detects that something is accessing it via the web interface. But we kinda want it to print nicely for the terminal. Above the dump() function, add header('Content-Type: cli'):

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

```
... lines 1 - 27
        if (!$form->isValid()) {
            header('Content-Type: cli');
            dump((string) $form->getErrors(true, false));die;
        }
... lines 32 - 145
```

That's a hack - but try the test now:

```
$ bin/phpunit -c app --filter testValidationErrors
```

Ok, that's a *sweet* looking dump. We've got the validation error for the nickname field and another for a missing CSRF token - we'll fix that soon. But, validation *is* working.

Collecting the Validation Errors

So now we just need to collect those errors and put them into a JSON response. To help with that, I'm going to paste a new private function into the bottom of ProgrammerController:

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

```
... lines 1 - 10
use Symfony\Component\Form\FormInterface;
... lines 12 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 151
    private function getErrorsFromForm(FormInterface $form)
    {
        $errors = array();
        foreach ($form->getErrors() as $error) {
            $errors[] = $error->getMessage();
        }

        foreach ($form->all() as $childForm) {
            if ($childForm instanceof FormInterface) {
                if ($childErrors = $this->getErrorsFromForm($childForm)) {
                    $errors[$childForm->getName()] = $childErrors;
                }
            }
        }

        return $errors;
    }
}
```

If you're coding with me, you'll find this in a code block on this page - copy it from there. Actually, I adapted this from some code in FOSRestBundle.

A Form object is a collection of other Form objects - one for each field. And sometimes, fields have sub-fields, which are yet *another* level of Form objects. It's a tree. And when validation runs, it attaches the errors to the Form object of the right field. That's the treky, I mean techy, explanation of this function: it recursively loops through that tree, fetching the errors off of each field to create an associative array of those errors.

Head back to newAction() and use this: $errors = $this->getErrorsFromForm() and pass it the $form object. Now, create a $data array that will eventually be our JSON response.

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

```
... lines 1 - 21
22      public function newAction(Request $request)
23      {
... lines 24 - 27
28          if (!$form->isValid()) {
29              $errors = $this->getErrorsFromForm($form);
30
31              $data = [
... lines 32 - 34
35              ];
... lines 36 - 37
38          }
... lines 39 - 53
54      }
... lines 55 - 170
```

Remember, we want type, title and errors keys. Add a type key: this is the machine name of what went wrong. How about validation_error - I'm making that up. For title - we'll have the human-readable version of what went wrong. Let's use: "There was a validation error". And for errors pass it the $errors array.

Finish it off! Return a new JsonResponse() with $data *and* the 400 status code:

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

```
... lines 1 - 21
22      public function newAction(Request $request)
23      {
... lines 24 - 27
28          if (!$form->isValid()) {
29              $errors = $this->getErrorsFromForm($form);
30
31              $data = [
32                  'type' => 'validation_error',
33                  'title' => 'There was a validation error',
34                  'errors' => $errors
35              ];
36
37              return new JsonResponse($data, 400);
38          }
... lines 39 - 53
54      }
... lines 55 - 170
```

Phew! Let's give it a try:

```
$ bin/phpunit -c app --filter testValidationErrors
```

Oof! That's not passing! That's a huge error. The dumped response looks perfect. The error started on ProgrammerControllerTest where we use assertResponsePropertiesExist(). Whoops! And there's the problem - I had assertResponsePropertyExists() - what you use to check for a single field. Make sure your's says assertResponsePropertiesExist().

Try it again:

```
$ bin/phpunit -c app --filter testValidationErrors
```

It's passing! Let's pretend I made that mistake on purpose - it was nice because we could see that the dumped response looks exactly like we wanted.

Chapter 3: PUT Validation and CSRF Tokens

Validation for newAction(), check! Now let's repeat for updateAction. And that's not much work - we just need to add the whole if (!$form->isValid()) block. I know you hate duplication, so copy the inside of that if statement, head to the bottom of the class, and add a new private function createValidationErrorResponse(). We'll pass it the $form object, and we should type-hint that argument with FormInterface because we're good programmers! Paste the stuff here:

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

```
... lines 1 - 10
use Symfony\Component\Form\FormInterface;
... lines 12 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 165
    private function createValidationErrorResponse(FormInterface $form)
    {
        $errors = $this->getErrorsFromForm($form);

        $data = [
            'type' => 'validation_error',
            'title' => 'There was a validation error',
            'errors' => $errors
        ];

        return new JsonResponse($data, 400);
    }
}
```

Cool! Any time we have a form, we can pass it here and get back a perfectly consistent validation error response. Go back up to newAction() and use this: return $this->createValidationErrorResponse() and pass it the $form object:

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

```
... lines 1 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 21
    public function newAction(Request $request)
    {
... lines 24 - 27
        if (!$form->isValid()) {
            return $this->createValidationErrorResponse($form);
        }
... lines 31 - 45
    }
... lines 47 - 177
}
```

Copy those three lines and repeat in updateAction():

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

```
... lines 1 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 88
    public function updateAction($nickname, Request $request)
    {
... lines 91 - 101
        $form = $this->createForm(new UpdateProgrammerType(), $programmer);
        $this->processForm($request, $form);

        if (!$form->isValid()) {
            return $this->createValidationErrorResponse($form);
        }
... lines 108 - 115
    }
... lines 117 - 177
}
```

We *could* write a test for this, but we've centralized everything so well, that I'm confident that if it works in newAction, it works in updateAction(). Basically, I think that's overkill. But we *should* re-run our test:

```
$ bin/phpunit -c app --filter testValidationErrors
```

All good. Now run *all* the tests:

```
$ bin/phpunit -c app
```

Oh! They break immediately! The POST is failing with a 400 response: invalid CSRF token - we saw this a few minutes ago. Every endpoint is failing because we're never sending a CSRF token.

Symfony forms *always* expect a token. But because we're building a stateless, or session-less API, we don't need CSRF tokens. You *would* need it if you have a JavaScript frontend that's relying on cookies to authenticate, but you don't need it if your API doesn't store the user in the session.

Let's turn it off. Inside ProgrammerType, in setDefaultOptions() - or configureOptions() if you're on a newer version of Symfony - set csrf_protection to false:

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

```
... lines 1 - 8
class ProgrammerType extends AbstractType
{
... lines 11 - 34
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
... lines 38 - 39
            'csrf_protection' => false,
        ));
    }
... lines 43 - 47
}
```

That'll do it! Try the tests:

```
$ bin/phpunit -c app
```

Back to green! If you're using your form types for HTML pages *and* on your API, you won't want to set csrf_protection to false

inside the class - that'll remove it everywhere. Instead, you can pass csrf_protection in as an option in the third argument to createForm() in your controller. Or you can do something fancier like a Form Type Extension and control this option on a global basis.

FOSRestBundle has an interesting version of this. In the View Layer part of their docs, they show a configuration option that disables CSRF protection based on a role the user has. The idea is that only users that are authenticated via the sessionless-API would have the role you put here. That's a cool idea.

Chapter 4: 99 api/problem+json(s)

Time for me to reveal why I chose this error response format with type, title and errors. Imagine if every JSON API returned the same format when things went wrong: always with these keys. That'd be pretty awesome. As API client, we'd always know what to expect and what things mean. That's a beautiful fairy tale.

In the real world, every API does whatever they want. But there are people out there working on standards for error responses, with the hope that someday, API's have some consistency.

The API Problem Details Format

One of those is called api problem details. Google for that. Ah, a boring spec document. Go ahead and read this whole thing, I'll wait... Kidding! I'll show you the good parts.

But first - click the draft ietf link. These drafts go through version, and this one has been replaced with a whole new document. And yea, these *are* just drafts - not official specs. But who cares? If a spec makes sense to you, why not follow it instead of making up your own format.

Scroll down to the example response. Two important things here. First, if you follow this spec, you should return a custom Content-type response header to advertise this: application/problem+json. When a client sees this - they can research it to find out what the fields in the response *mean* in human terms.

Second, check out the fields: the main ones are type and title. type is a unique string for *what* went wrong. It's supposed to be a URL - our's is just a key. We'll revisit that later. Next, this says title is a human-readable version of type and there are a bunch of other optional fields. The Extension Members section says that you can also add whatever other fields you want. We're adding an extra errors key.

Adding the application/problem+json Header

So we're already following this format - or at least we're pretty close. So I want to advertise this to our clients so they can dig into what each key means. Copy the application/problem+json Content-Type header so we can use it.

First, check for this in the test: $this->assertEquals() with application/problem+json as the expected value and $response->getHeader('Content-Type') for the actual value:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 123
    public function testValidationErrors()
    {
... lines 126 - 144
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type'));
    }
}
```

Make sure it fails - run *just* this test:

```
$ bin/phpunit -c app --filter testValidationErrors
```

Yep, it fails. Now go to ProgrammerController and find createValidationErrorResponse. Instead of returning JsonResponse, set it to a $response variable and then call $response->headers->set() with Content-Type and application/problem+json. Return this:

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

```
... lines 1 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 165
    private function createValidationErrorResponse(FormInterface $form)
    {
... lines 168 - 175
        $response = new JsonResponse($data, 400);
        $response->headers->set('Content-Type', 'application/problem+json');

        return $response;
    }
}
```

See if this fixed the test:

```
$ bin/phpunit -c app --filter testValidationErrors
```

Error!

```
Attempted to call function JsonResponse from namespace "AppBundle\Controller\Api"
```

That looks like a "Ryan" mistake - I deleted the new keyword before JsonResponse. You probably saw me do that. Put it back and *now* try the tests:

Beautiful green! And now we're advertising our special error format.

Chapter 5: Modeling the Error: ApiProblem Class

Ok, we've got a format for errors - and we're going to use this whenever *anything* goes wrong - like a 404 error, authentication error, 500 error, whatever. And each time, this format needs to be *perfectly* consistent.

So instead of creating this $data array by hand when things go wrong, let's create a class that models all this stuff.

The ApiProblem Class

I actually started this for us. In PhpStorm, I'll switch my view back so I can see the resources/ directory at the root. Copy the ApiProblem.php file. In AppBundle, create a new Api directory and paste the file here:

47 lines | src/AppBundle/Api/ApiProblem.php

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

/**
 * A wrapper for holding data to be used for a application/problem+json response
 */
class ApiProblem
{
    private $statusCode;

    private $type;

    private $title;

    private $extraData = array();

    public function __construct($statusCode, $type, $title)
    {
        $this->statusCode = $statusCode;
        $this->type = $type;
        $this->title = $title;
    }
... lines 24 - 45
}
```

The namespace is already AppBundle\Api - so that's perfect. This holds data for an application/problem+json response. It has properties for type, title and statusCode - these being the three *main* fields from the spec.

And it also has a spot for extra fields:

47 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
... lines 10 - 15
    private $extraData = array();
... lines 17 - 36
    public function set($name, $value)
    {
        $this->extraData[$name] = $value;
    }
... lines 41 - 45
}
```

If you call set(), we can add any extra stuff, like the errors key for validation. And when we're all done, we'll call the toArray() method to get all this back as a flat, associative array:

47 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
... lines 10 - 24
    public function toArray()
    {
        return array_merge(
            $this->extraData,
            array(
                'status' => $this->statusCode,
                'type' => $this->type,
                'title' => $this->title,
            )
        );
    }
... lines 36 - 45
}
```

Using ApiProblem

Let's use this back in ProgrammerController. Start with $apiProblem = new ApiProblem(). The status code is 400, the type is validation_error and the title is There was a validation error. Let's knock this onto multiple lines for readability:

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

```
... lines 1 - 16
class ProgrammerController extends BaseController
{
... lines 19 - 166
    private function createValidationErrorResponse(FormInterface $form)
    {
... lines 169 - 170
        $apiProblem = new ApiProblem(
            400,
            'validation_error',
            'There was a validation error'
        );
... lines 176 - 181
    }
}
```

Get rid of the $data variable. To add the extra errors field, call $apiProblem->set() and pass it the errors string and the $errors variable:

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

```
... lines 1 - 166
    private function createValidationErrorResponse(FormInterface $form)
    {
... lines 169 - 170
        $apiProblem = new ApiProblem(
            400,
            'validation_error',
            'There was a validation error'
        );
        $apiProblem->set('errors', $errors);
... lines 177 - 181
    }
... lines 183 - 184
```

The last step is to update JsonResponse. Instead of $data, use $apiProblem->toArray(). And to avoid duplication, use $apiProblem->getStatusCode() instead of 400:

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

```
... lines 1 - 166
    private function createValidationErrorResponse(FormInterface $form)
    {
... lines 169 - 170
        $apiProblem = new ApiProblem(
            400,
            'validation_error',
            'There was a validation error'
        );
        $apiProblem->set('errors', $errors);

        $response = new JsonResponse($apiProblem->toArray(), $apiProblem->getStatusCode());
... lines 179 - 180
        return $response;
    }
... lines 183 - 184
```

It's not perfect yet - but this is a lot more dependable. Nothing should have change - so try the tests:

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

And yep! We're still green.

But go back and make the test fail somehow - like change the assert for the header. I want to see the response for myself. Re-run things:

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

Scroll up to the dumped response. Yes - we've got the Content-Type header, the type and title keys, *and* a new status field that the spec recommends.

Fix that test. Ok, now we're ready for other stuff to go wrong.

Chapter 6: Keeping Problem types Consistent

Look back at the title field in the spec:

> A short, human-readable summary of the problem type. It SHOULD NOT change from occurrence to occurrence of the problem, except if you're translating it.

In human terms, this means that *every* time we have a validation_error type, the title should be exactly There was a validation error. So when we're validating in other places in the future, both of the type *and* the title need to be *exactly* the same. Otherwise, our API clients are going to wonder what kind of inconsistent jerk programmed this API.

Because validation_error is now a "special string" in the app, I think this is great spot for a constant. In ApiProblem, add a constant called TYPE_VALIDATION_ERROR and set it to the string:

49 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
    const TYPE_VALIDATION_ERROR = 'validation_error';
... lines 11 - 47
}
```

Ok, use that back in the controller: ApiProblem::TYPE_VALIDATION_ERROR:

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

```
... lines 1 - 16
class ProgrammerController extends BaseController
{
... lines 19 - 166
    private function createValidationErrorResponse(FormInterface $form)
    {
... lines 169 - 170
        $apiProblem = new ApiProblem(
            400,
            ApiProblem::TYPE_VALIDATION_ERROR,
            'There was a validation error'
        );
... lines 176 - 181
    }
}
```

Ok, that's better. Next, I also don't want to mess up the title. Heck, I don't even want to have to *write* the title anywhere - can't it be guessed based on the type?

I think so - let's just create a map from the type, to its associated title. In ApiProblem, add a private static $titles property. Let's make it an associative array map: from TYPE_VALIDATION_ERROR to its message: There was a validation error:

58 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
    const TYPE_VALIDATION_ERROR = 'validation_error';

    private static $titles = array(
        self::TYPE_VALIDATION_ERROR => 'There was a validation error'
    );
... lines 15 - 56
}
```

In __construct(), let's kill the $title argument completely. Instead - just use self::$titles and look it up with $type:

58 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
... lines 10 - 11
    private static $titles = array(
        self::TYPE_VALIDATION_ERROR => 'There was a validation error'
    );
... lines 15 - 23
    public function __construct($statusCode, $type)
    {
... lines 26 - 32
        $this->title = self::$titles[$type];
    }
... lines 35 - 56
}
```

And we should code defensively, in case we mess something up later. Check if (!isset(self::$titles[$type])) and throw a huge Exception message to our future selves. How about, "Hey - buddy, use your head!". Or, more helpfully: "No title for type" and pass in the value.

58 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 23
    public function __construct($statusCode, $type)
    {
... lines 26 - 28
        if (!isset(self::$titles[$type])) {
            throw new \InvalidArgumentException('No title for type '.$type);
        }

        $this->title = self::$titles[$type];
    }
... lines 35 - 58
```

Now we can pop off the last argument in the controller:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 147
    public function testInvalidJson()
    {
... lines 150 - 162
    }
... line 164
}
```

Ok future me, good lucking screwing up our API problem responses. This is easy to use, and hard to break. Time to re-run the test:

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

Great! Let's keep hardening our API.

Chapter 7: The All-Important HttpExceptionInterface

What do you think would happen if we POST'ed some badly-formatted JSON to an endpoint? Because, I'm not really sure - but I bet the error wouldn't be too obvious to the client.

The invalidJson Test

Let's add a test for this and think about how we *want* our API to act if someone mucks up their JSON. Copy testValidationErrors() - it should be pretty similar. Name the new method testInvalidJSON():

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 147
    public function testInvalidJson()
    {
... lines 150 - 162
    }
... line 164
}
```

And we can't use this $data array anymore, json_encode is too good at creating *valid* JSON. Overachiever. Replace it with an $invalidJson variable - we'll have to create really bad JSON ourselves. Let's see here, start with one piece of valid JSON, remove a comma and liven things up with a hanging quotation mark, and that oughta do it! Now pass $invalidJson as the request body:

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

```
... lines 1 - 147
    public function testInvalidJson()
    {
        $invalidBody = <<<EOF
{
    "avatarNumber" : "2
    "tagLine": "I'm from a test!"
}
EOF;

        $response = $this->client->post('/api/programmers', [
            'body' => $invalidBody
        ]);
... lines 160 - 162
    }
... lines 164 - 166
```

Ok, time to think about how we want the response to look. The 400 status code is good. Invalid JSON is the client's fault, and any status code starting with 4 is for when *they* mess up. You could also use 422 - Unprocessable Entity - if you want to enhance your nerdery. But, nobody is going to notice.

And since we're curious about how our API *currently* handles invalid JSON, use $this->debugResponse() right above the assert:

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

```
... lines 1 - 147
148     public function testInvalidJson()
149     {
... lines 150 - 156
157         $response = $this->client->post('/api/programmers', [
158             'body' => $invalidBody
159         ]);
160         $this->debugResponse($response);
161 
162         $this->assertEquals(400, $response->getStatusCode());
163     }
... lines 164 - 166
```

Copy the test name and give it a try:

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

Cool - the test passes with a 400 response, but the error isn't about having invalid JSON. Instead, it looks like we're missing our nickname. Ok, so let's add a nickname:

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

```
... lines 1 - 147
148     public function testInvalidJson()
149     {
150         $invalidBody = <<<EOF
151 {
152     "nickname": "JohnnyRobot",
153     "avatarNumber" : "2
154     "tagLine": "I'm from a test!"
155 }
156 EOF;
157 
158         $response = $this->client->post('/api/programmers', [
159             'body' => $invalidBody
160         ]);
... lines 161 - 163
164     }
... lines 165 - 167
```

And try the test again:

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

And we *still* fail because the nickname field is missing. So apparently, if we send invalid JSON, it acts like we're sending nothing. So good luck to any future API client trying to debug this.

Handling Invalid JSON

In ProgrammerController, search for json_decode - you'll find it in processForm():

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

```
... lines 1 - 17
18  class ProgrammerController extends BaseController
19  {
... lines 20 - 141
142     private function processForm(Request $request, FormInterface $form)
143     {
144         $data = json_decode($request->getContent(), true);
... lines 145 - 150
151     }
... lines 152 - 185
186 }
```

If the $body has a bad format, then $data will be null. Add an if to test for that: if null === $data then we need to return that 400 status code:

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

```
... lines 1 - 141
142     private function processForm(Request $request, FormInterface $form)
143     {
144         $data = json_decode($request->getContent(), true);
145         if ($data === null) {
... line 146
147         }
... lines 148 - 150
151     }
... lines 152 - 187
```

The HttpException(Interface)

But wait! There's a huge problem! When processForm() is called, its return value isn't used:

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

```
... lines 1 - 17
18  class ProgrammerController extends BaseController
19  {
... lines 20 - 23
24      public function newAction(Request $request)
25      {
... lines 26 - 27
28          $this->processForm($request, $form);
... lines 29 - 47
48      }
... lines 49 - 90
91      public function updateAction($nickname, Request $request)
92      {
... lines 93 - 104
105         $this->processForm($request, $form);
... lines 106 - 117
118     }
... lines 119 - 185
186 }
```

So if we return a Response from processForm()... good for us! Nobody will actually do anything with that: newAction will continue on like normal. The only way we can break the flow from inside processForm() is by throwing an exception. But as you're probably thinking, if you throw an exception in Symfony, that turns into a *500* error. We need a 400 error.

And it turns out, that's totally possible - and it's a really important concept for API's. First, I just said that throwing an exception causes a 500 error in Symfony. That's just not the whole story. Throw a new HttpException from HttpKernel. It has 2 arguments: the status code - 400 - and a message - just "Invalid JSON" for now. Don't worry *yet* about returning our nice API problem JSON:

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

```
... lines 1 - 141
142     private function processForm(Request $request, FormInterface $form)
143     {
144         $data = json_decode($request->getContent(), true);
145         if ($data === null) {
146             throw new HttpException(400, 'Invalid JSON body!');
147         }
... lines 148 - 150
151     }
... lines 152 - 187
```

So here's the truth about exceptions: any exception will turn into a 500 error, *unless* that exception implements the HttpExceptionInterface:

35 lines | vendor/symfony/symfony/src/Symfony/Component/HttpKernel/Exception/HttpExceptionInterface.php

```
... lines 1 - 11
12  namespace Symfony\Component\HttpKernel\Exception;
... lines 13 - 18
19  interface HttpExceptionInterface
20  {
... lines 21 - 25
26      public function getStatusCode();
... lines 27 - 32
33      public function getHeaders();
34  }
```

It has two functions: getStatusCode() and getHeaders(). The HttpException class we're throwing implements this. That means we can throw this from *anywhere* in our code, stop the flow, but control the status code of the response. Symfony ships with a bunch of convenience sub-classes for common status codes, like ConflictHttpException, which automatically gives you a 409 status code. All of those classes are optional: you could use HttpException for everything.

Ok, back to reality. Since we're throwing this, the response should still be 400, so the test should still pass. But, instead of getting back a validation error, we should get back our simple "Invalid JSON" text message:

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

Yep! It passes and prints out "Invalid JSON". The "There was an error" part is from my test helper - but the red text below is the actual response. The point is that we *are* handling invalid JSON now, but we're not sending back the awesome API Problem JSON format yet.

Chapter 8: Creating the Invalid JSON ApiProblem, and then...

The nice API Problem JSON format always has a type key, so let's at least start looking for that in the response. Use $this->asserter()->assertResponsePropertyEquals() and pass it the $response and type as the key. For the value - how about invalid_body_format. That's our *second* special error "type" - the first was validation_error.

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 147
    public function testInvalidJson()
    {
... lines 150 - 162
        $this->assertEquals(400, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyEquals($response, 'type', 'invalid_body_format');
    }
... line 166
}
```

This should get our test to fail again:

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

Gooood - we're still returning the exception HTML.

Creating the ApiProblem

Let's fix this just like we did for validation errors: by creating an ApiProblem object. First, we need a new type constant. Create a second constant called TYPE_INVALID_REQUEST_BODY_FORMAT and set that to the string from our test: invalid_body_format. Setup a title for this too: how about "Invalid JSON format sent". And I better fix my syntax error:

60 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
    const TYPE_VALIDATION_ERROR = 'validation_error';
    const TYPE_INVALID_REQUEST_BODY_FORMAT = 'invalid_body_format';

    private static $titles = array(
        self::TYPE_VALIDATION_ERROR => 'There was a validation error',
        self::TYPE_INVALID_REQUEST_BODY_FORMAT => 'Invalid JSON format sent',
    );
... lines 17 - 58
}
```

Back in ProgrammerController, we can get to work: $apiProblem = new ApiProblem(). Pass it the 400 status code and the type: ApiProblem::TYPE_INVALID_REQUEST_BODY_FORMAT:

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

```
... lines 1 - 17
class ProgrammerController extends BaseController
{
... lines 20 - 141
    private 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);
... lines 147 - 148
        }
... lines 150 - 152
    }
... lines 154 - 187
}
```

We're stuck

Gosh, everything is going really well! And now we're stuck. For validation, we took the ApiProblem, turned it into a Response and returned it. But inside processForm(), we're big trouble: the return value of this method isn't being used. We can only *throw* an exception to stop the flow. And while we *can* control the status code of an exception, the response body that an HttpException generates is still an HTML error page.

Chapter 9: ApiProblemException

The ApiProblem object knows everything about how the response should look, including the status code and the response body information. So, it'd be great to have an easy way to convert this into a Response.

But, I want to go further. Sometimes, having a Response isn't enough. Like in processForm(): since nothing uses its return value. So the only way to break the flow is by throwing an exception.

Here's the goal: create a special exception class, pass it the ApiProblem object, and then have some central layer convert that into our beautiful API problem JSON formatted response. So whenever something goes wrong, we'll just need to create the ApiProblem object and then throw this special exception. That'll be it, in *any* situation.

Create the ApiProblemException

In the Api directory, create a new class called ApiProblemException. Make this extend HttpException - because I like that ability to set the status code on this:

10 lines | src/AppBundle/Api/ApiProblemException.php

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

use Symfony\Component\HttpKernel\Exception\HttpException;

class ApiProblemException extends HttpException
{
}
```

Next, we need to be able to attach an ApiProblem object to this exception class, so that we have access to it later when we handle all of this. Let's pass this via the constructor. Use cmd+n - or go to the "Generate" menu at the top - and override the __construct method. Now, add ApiProblem $apiProblem as the first argument. Also create an $apiProblem property and set this there:

18 lines | src/AppBundle/Api/ApiProblemException.php

```
... lines 1 - 6
class ApiProblemException extends HttpException
{
    private $apiProblem;

    public function __construct(ApiProblem $apiProblem, $statusCode, $message = null, \Exception $previous = null, array $headers =
    {
        $this->apiProblem = $apiProblem;

        parent::__construct($statusCode, $message, $previous, $headers, $code);
    }
}
```

This won't do *anything* special yet: this is still just an HttpException that happens to have an ApiProblem attached to it.

Back in ProgrammerController, we can start using this. Throw a new ApiProblemException. Pass it $apiProblem as the first argument and 400 next:

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

```
... lines 1 - 142
    private 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,
                400
            );
        }
... lines 154 - 156
    }
... lines 158 - 193
```

Run the test:

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

It still acts like before: with a 400 status code, and now an exception with no message.

Simplifying the ApiProblemException Constructor

Before we handle this, we can make one minor improvement. Remove the $statusCode and $message arguments because we can get those from the ApiProblem itself. Replace that with $statusCode = $apiProblem->getStatusCode(). And I just realized I messed up my first line - make sure you have $this->apiProblem = $apiProblem. Also add $message = $apiProblem->getTitle():

20 lines | src/AppBundle/Api/ApiProblemException.php

```
... lines 1 - 6
class ApiProblemException extends HttpException
{
... lines 9 - 10
    public function __construct(ApiProblem $apiProblem, \Exception $previous = null, array $headers = array(), $code = 0)
    {
        $this->apiProblem = $apiProblem;
        $statusCode = $apiProblem->getStatusCode();
        $message = $apiProblem->getTitle();

        parent::__construct($statusCode, $message, $previous, $headers, $code);
    }
}
```

Hey wait! ApiProblem doesn't have a getTitle() method yet. Ok, let's go add one. Use the Generate menu again, select "Getters" and choose title:

65 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 7
class ApiProblem
{
... lines 10 - 59
    public function getTitle()
    {
        return $this->title;
    }
}
```

In ProgrammerController, simplify this:

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

```
... lines 1 - 142
    private 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);
        }
... lines 151 - 153
    }
... lines 155 - 190
```

It'll figure out the status code and message for us.

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

The exception class is perfect - we just need to add that central layer that'll convert this into the beautiful API Problem JSON response. Instead of this HTML stuff.

Chapter 10: Request Format: Why Exceptions Return HTML

When you throw an exception in Symfony - even an HttpException - it returns an HTML page. Notice the Content-Type header here of text/html. And in reality, this is returning a full, giant HTML exception page - my test helpers are just summarizing things.

Why is that? Why does Symfony default to the idea that if something goes wrong, it should return HTML?

Request Format

Here's the answer: for every single request, Symfony has what's called a "request format", and it defaults to html. But there are a number of different ways to say "Hey Symfony, the user wants *json*, so if something goes wrong, give them that".

The easiest way to set the request format is in your routing. Open up app/config/routing.yml:

8 lines | app/config/routing.yml

```
... lines 1 - 4
app_api:
    resource: "@AppBundle/Controller/Api"
    type:     annotation
```

When we import the routes from our API controllers, we want *all* of them to have a json request format. To do that, add a defaults key. Below that, set a magic key called _format to json:

10 lines | app/config/routing.yml

```
... lines 1 - 4
app_api:
    resource: "@AppBundle/Controller/Api"
    type:     annotation
    defaults:
        _format: json
```

For us, this is optional, because in a minute, we're going to *completely* take control of exceptions for our API. But with just this, re-run the tests:

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

Yes! Now we get a Content-Type header of application/json and because we're in the dev environment, it returns the full stack trace as JSON.

This is cool. But the JSON structure still won't be right. So let's take full control using our ApiProblemException.

Chapter 11: Global RESTful Exception Handling

When we throw an ApiProblemException, we need our app to automatically turn that into a nicely-formatted API Problem JSON response and return it. That code will look like what we have down here for validation, but it needs to live in a global spot.

Whenever an exception is thrown in Symfony, it dispatches an event called kernel.exception. If we attach a listener function to that event, we can take full control of how exceptions are handled. If creating a listener is new to you, we have a chapter on that in our Journey series called Interrupt Symfony with an Event Subscriber.

In AppBundle, create an EventListener directory. Add a new class in here called ApiExceptionSubscriber and make sure it's in the AppBundle\EventListener namespace:

8 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

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

class ApiExceptionSubscriber
{
}
```

There are two ways to hook into an event: via a listener or a subscriber. They're really the same thing, but I think subscribers are cooler. To hook one up, make this class implement EventSubscriberInterface - the one from Symfony. Now, hit cmd+n - or go to the the Code->Generate menu - select "Implement Methods" and select getSubscribedEvents. That's a fast way to generate the *one* method from EventSubscriberInterface that we need to fill in:

14 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 4
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class ApiExceptionSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        // TODO: Implement getSubscribedEvents() method.
    }
}
```

Return an array with one just entry. The key is the event's name - use KernelEvents::EXCEPTION - that's really just the string kernel.exception. Assign that to the string: onKernelException. That'll be the name of our method in this class that should be called whenever an exception is thrown. Create that method: public function onKernelException():

22 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 5
use Symfony\Component\HttpKernel\KernelEvents;
... line 7
class ApiExceptionSubscriber implements EventSubscriberInterface
{
    public function onKernelException()
    {

    }

    public static function getSubscribedEvents()
    {
        return array(
            KernelEvents::EXCEPTION => 'onKernelException'
        );
    }
}
```

So once we tell Symfony about this class, whenever an exception is thrown, Symfony will call this method. And when it does, it'll pass us an $event argument object. But what type of object is that? Hold cmd - or control for Windows and Linux - and click the EXCEPTION constant. The documentation *above* it tells us that we'll be passed a GetResponseForExceptionEvent object. Close that class and type-hint the event argument. Don't forget your use statement:

23 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 5
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
... lines 7 - 8
class ApiExceptionSubscriber implements EventSubscriberInterface
{
    public function onKernelException(GetResponseForExceptionEvent $event)
    {

    }
... lines 15 - 21
}
```

The Subscriber Logic

Listeners to kernel.exception have one big job: to *try* to understand what went wrong and return a Response for the error. The big exception page we see in dev mode is caused by a core Symfony listener to this same event. We throw an exception and it gives us the pretty exception page.

So our missing is clear: detect if an ApiProblemException was thrown and create a nice Api Problem JSON response if it was.

First, to get access to the exception that was just thrown, call getException() on the $event. So far, we *only* want our listener to act if this is an ApiProblemException object. Add an if statment: if !$e instanceof ApiProblemException, then just return immediately:

27 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 11
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
        $e = $event->getException();
        if (!$e instanceof ApiProblemException) {
            return;
        }
    }
... lines 19 - 27
```

For now, that'll mean that normal exceptions will still be handled via Symfony's core listener.

Now that we know this is an ApiProblemException, let's turn it into a Response. Go steal the last few lines of the validation response code from ProgrammerController. Put this inside onKernelException(). You'll need to add the use statement for JsonResponse manually:

38 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 12
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
        $e = $event->getException();
        if (!$e instanceof ApiProblemException) {
            return;
        }
... lines 19 - 21
        $response = new JsonResponse(
            $apiProblem->toArray(),
            $apiProblem->getStatusCode()
        );
        $response->headers->set('Content-Type', 'application/problem+json');
... lines 27 - 28
    }
... lines 30 - 38
```

But we don't have an $apiProblem variable yet. There *is* an ApiProblem object inside the ApiProblemException as a property, but we don't have a way to access it yet. Go back to the Generate menu - select Getters - and choose the apiProblem property:

25 lines | src/AppBundle/Api/ApiProblemException.php

```
... lines 1 - 6
class ApiProblemException extends HttpException
{
... lines 9 - 19
    public function getApiProblem()
    {
        return $this->apiProblem;
    }
}
```

In the subscriber, we can say $apiProblem = $e->getApiProblem():

38 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 10
class ApiExceptionSubscriber implements EventSubscriberInterface
{
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 15 - 19
        $apiProblem = $e->getApiProblem();

        $response = new JsonResponse(
            $apiProblem->toArray(),
            $apiProblem->getStatusCode()
        );
        $response->headers->set('Content-Type', 'application/problem+json');
... lines 27 - 28
    }
... lines 30 - 36
}
```

This is now *exactly* the Response we want to send back to the client. To tell Symfony to use this, call $event->setResponse() and pass it the $response:

38 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 12
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 15 - 27
        $event->setResponse($response);
    }
... lines 30 - 38
```

Registering the Event Subscriber

There's just one more step left: telling Symfony about the subscriber. Go to app/config/services.yml. Give the service a name - how about api_problem_subscriber. Then fill in the class with ApiExceptionSubscriber and give it an empty arguments key. The secret to telling Symfony that this service is an event subscriber is with a tag named kernel.event_subscriber:

24 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 19
    api_exception_subscriber:
        class: AppBundle\EventListener\ApiExceptionSubscriber
        arguments: []
        tags:
            - { name: kernel.event_subscriber }
```

That tag is enough to tell Symfony about our subscriber - it'll take care of the rest.

Head back to our test where we send invalid JSON and expect the 400 status code. This already worked before, but the response was HTML, so the next assert - for a JSON response with a type property - has been failing hard. Actually, I totally messed up that assert earlier - make sure you're asserting a type key, not test:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 147
    public function testInvalidJson()
    {
... lines 150 - 162
        $this->asserter()->assertResponsePropertyEquals($response, 'type', 'invalid_body_format');
    }

}
```

So, type should be set to invalid_body_format because the ApiProblem has that type set via the constant:

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 142
    private 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);
        }
... lines 151 - 153
    }
... lines 155 - 187
    }
}
```

But with the exception subscriber in place, we *should* now get the JSON response we want. Ok, moment of truth:

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

It passes! This is *huge*! We now have a central way for triggering and handling errors. Take out the debugResponse() call.

Celebrate by throwing an ApiProblemException for validations errors too. Replace all the Response-creation logic in createValidationErrorResponse() with a simple throw new ApiProblemException() and pass it the $apiProblem:

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 173
    private function createValidationErrorResponse(FormInterface $form)
    {
... lines 176 - 183
        throw new ApiProblemException($apiProblem);
    }
}
```

That's all the code we need now, no matter where we are. And now, the method name - createValidationErrorResponse() isn't really accurate. Change it to throwApiProblemValidationException():

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 173
    private function throwApiProblemValidationException(FormInterface $form)
    {
... lines 176 - 183
        throw new ApiProblemException($apiProblem);
    }
}
```

Search for the 2 spots that use that and update the name. And we don't need to have a return statement anymore: just call the function and it'll throw the exception for us:

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

```
... lines 1 - 24
    public function newAction(Request $request)
    {
... lines 27 - 30
        if (!$form->isValid()) {
            $this->throwApiProblemValidationException($form);
        }
... lines 34 - 48
    }
... lines 50 - 91
    public function updateAction($nickname, Request $request)
    {
... lines 94 - 107
        if (!$form->isValid()) {
            $this->throwApiProblemValidationException($form);
        }
... lines 111 - 118
    }
... lines 120 - 185
}
```

Re-test *everything*:

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

Now we're green and we can send back exciting error responses from anywhere in our code. But what about other exceptions, like 404 exceptions?

Chapter 12: Handling 404's + other Errors

What should the structure of a 404 response from our API look like? It's obvious: we'll want to return that same API Problem JSON response format. We want to return this whenever *anything* goes wrong.

Planning the Response

Start by planning out how the 404 should look with a new test method - test404Exception. Let's make a GET request to /api/programmers/fake and assert the easy part: that the status code is 404. We also know that we want the nice application/problem+json Content-Type header, so assert that too:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 165
    public function test404Exception()
    {
        $response = $this->client->get('/api/programmers/fake');

        $this->assertEquals(404, $response->getStatusCode());
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type'));
... lines 172 - 173
    }
}
```

We know the JSON will at least have type and title properties. So what would be good values for those? This is a weird situation. Usually, type conveys *what* happened. But in this case, the 404 status code already says everything we need to. Using some type value like not_found would be fine, but totally redundant.

Look back at the Problem Details Spec. Under "Pre-Defined Problem Types", it says that if the status code is enough, you can set type to about:blank. And when you do this, it says that we should set title to whatever the standard text is for that status code. A 404 would be "Not Found".

Add this to the test: use $this->asserter()->assertResponsePropertyEquals() to assert that type is about:blank. And do this all again to assert that title is Not Found:

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

```
... lines 1 - 165
    public function test404Exception()
    {
        $response = $this->client->get('/api/programmers/fake');

        $this->assertEquals(404, $response->getStatusCode());
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type'));
        $this->asserter()->assertResponsePropertyEquals($response, 'type', 'about:blank');
        $this->asserter()->assertResponsePropertyEquals($response, 'title', 'Not Found');
    }
... lines 175 - 176
```

How 404's Work

A 404 happens whenever we call $this->createNotFoundException() in a controller. If you hold cmd or ctrl and click that method, you'll see that this is just a shortcut to throw a special NotFoundHttpException. And *all* of the other errors that might happen will ultimately just be different exceptions being thrown from different parts of our app.

The only thing that makes *this* exception special is that it extends that very-important HttpException class. That's why throwing this causes a 404 response. But otherwise, it's equally as exciting as any other exception.

Handling *all* Errors

In ApiExceptionSubscriber, we're only handling ApiException's so far. But if we handled *all* exceptions, we could turn *everything* into the nice format we want.

Reverse the logic on the if statement and set the $apiProblem variable inside:

45 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 12
class ApiExceptionSubscriber implements EventSubscriberInterface
{
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
        $e = $event->getException();

        if ($e instanceof ApiProblemException) {
            $apiProblem = $e->getApiProblem();
        } else {
... lines 22 - 26
        }
... lines 28 - 35
    }
... lines 37 - 43
}
```

Add an else. In all other cases, we'll need to create the ApiProblem ourselves. The first thing we need to figure out is what status code this exception should have. Create a $statusCode variable. Here, check if $e is an instanceof HttpExceptionInterface: that special interface that lets an exception control its status code. So if it is, set the status code to $e->getStatusCode(). Otherwise, we have to assume that it's 500:

45 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 14
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 17 - 18
        if ($e instanceof ApiProblemException) {
... line 20
        } else {
            $statusCode = $e instanceof HttpExceptionInterface ? $e->getStatusCode() : 500;
... lines 23 - 26
        }
... lines 28 - 35
    }
... lines 37 - 45
```

Now use this to create an ApiProblem: $apiProblem = new ApiProblem() and pass it the $statusCode:

45 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 14
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 17 - 18
        if ($e instanceof ApiProblemException) {
... line 20
        } else {
            $statusCode = $e instanceof HttpExceptionInterface ? $e->getStatusCode() : 500;

            $apiProblem = new ApiProblem(
                $statusCode
            );
        }
... lines 28 - 35
    }
... lines 37 - 45
```

For the type argument, we *could* pass about:blank - that *is* what we want. But then in ApiProblem, we'll need a constant for this, and that constant will need to be mapped to a title. But we actually want the title to be dynamic based on whatever the status code is: 404 is "Not Found", 403 is "Forbidden", etc. So, don't pass *anything* for the type argument. Let's handle all of this logic inside ApiProblem itself.

In there, start by making the $type argument optional:

77 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 8
class ApiProblem
{
... lines 11 - 26
    public function __construct($statusCode, $type = null)
    {
... lines 29 - 47
    }
... lines 49 - 75
}
```

And *if* $type is exactly null, then set it to about:blank. Make sure the $this->type = $type assignment happens *after* all of this:

77 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 26
    public function __construct($statusCode, $type = null)
    {
... lines 29 - 30
        if ($type === null) {
            // no type? The default is about:blank and the title should
            // be the standard status code message
            $type = 'about:blank';
... lines 35 - 43
        }

        $this->type = $type;
... line 47
    }
... lines 49 - 77
```

For $title, we just need a map from the status code to its official description. Go to Navigate->Class - that's cmd+o on a Mac.

Look for Response and open the one inside HttpFoundation. It has a really handy public $statusTexts map that's exactly what we want:

1276 lines | vendor/symfony/symfony/src/Symfony/Component/HttpFoundation/Response.php

```
... lines 1 - 11
12  namespace Symfony\Component\HttpFoundation;
... lines 13 - 20
21  class Response
... lines 22 - 124
125     public static $statusTexts = array(
... lines 126 - 150
151         403 => 'Forbidden',
152         404 => 'Not Found',
... lines 153 - 185
186     );
... lines 187 - 1274
1275 }
```

Set the $title variable - but use some if logic in case we have some weird status code for some reason. If it *is* in the $statusTexts array, use it. Otherwise, well, this is kind of a weird situation. Use Unknown Status Code with a frowny face:

77 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 26
27     public function __construct($statusCode, $type = null)
28     {
... lines 29 - 30
31         if ($type === null) {
... lines 32 - 33
34             $type = 'about:blank';
35             $title = isset(Response::$statusTexts[$statusCode])
36                 ? Response::$statusTexts[$statusCode]
37                 : 'Unknown status code :(';
... lines 38 - 43
44         }
... lines 45 - 47
48     }
... lines 49 - 77
```

If the $type *is* set - we're in the normal case. Move the check up there and add $title = self::$titles[$type]. After everything, assign $this->title = $title:

77 lines | src/AppBundle/Api/ApiProblem.php

```
... lines 1 - 26
    public function __construct($statusCode, $type = null)
    {
... lines 29 - 30
        if ($type === null) {
... lines 32 - 37
        } else {
            if (!isset(self::$titles[$type])) {
                throw new \InvalidArgumentException('No title for type '.$type);
            }

            $title = self::$titles[$type];
        }

        $this->type = $type;
        $this->title = $title;
    }
... lines 49 - 77
```

Now the code we wrote in ApiExceptionSubscriber should work: a missing $type tells ApiProblem to use all the about:blank stuff. Time to try this: copy the test method name, then run:

```
$ ./bin/phpunit -c app --filter test404Exception
```

Aaaand that's green. It's so nice when things work.

What we just did is *huge*. If a 404 exception is thrown *anywhere* in the system, it'll map to the nice Api Problem format we want. In fact, if *any* exception is thrown it ends up with that format. So if your database blows, an exception is thrown. Sure, that'll map to a 500 status code, but the JSON format will be just like every other error.

Chapter 13: The Helpful Detail Key

Our end goal is to make our API easy to use so if something goes wrong our clients can actually debug it without pulling their hair out or having to email us.

Whenever you throw an exception in PHP there is going to be a message, like "No programmer found for username":

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 54
    public function showAction($nickname)
    {
        $programmer = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findOneByNickname($nickname);

        if (!$programmer) {
            throw $this->createNotFoundException(sprintf(
                'No programmer found with nickname "%s"',
                $nickname
            ));
        }
... lines 67 - 70
    }
... lines 72 - 185
}
```

This message is usually just for us as developers. When we're in development mode *we* see this message, but our clients don't. But, sometimes this message is useful, like in this case. Having something in the response that says "No programmer found for username" would help me as a client know that I have the right URL but that nickname I'm trying to use is missing.

There are other cases where we don't want to show the exception message. For example, if our database credentials are incorrect and we're getting a 500 error, we don't want to tell our client "invalid database credentials" -- that is a detail to hide.

Introducing the detail Property

Back in the spec, do we have a field for this? It's not supposed to be title, because that's supposed to be the same for every type. We could always add our own but if you look there is something called "detail" which is a "human readable explanation specific to *this* occurence of the problem." That's perfect for our use case!

Back in ProgrammerControllerTest let's look for this exact message, "No programmer found for username".

At the bottom we'll say $this->asserter()->assertResponsePropertyEquals() we'll fill this in so that when there's a 404 there will be a detail field and it should be set to "No programmer found for username fake" because that's what's in the URL:

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

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 165
    public function test404Exception()
    {
        $response = $this->client->get('/api/programmers/fake');
... lines 169 - 173
        $this->asserter()->assertResponsePropertyEquals($response, 'detail', 'No programmer found with nickname "fake"');
    }
}
```

And if we try this out in our terminal, it's failing nice:

```
$ ./bin/phpunit -c app --filter test404Exception
```

There's no detail property yet. But no worries, creating that is easy! It all happens inside of our ApiExceptionSubscriber.

Adding Exception Message as detail

Very simply, we say $apiProblem->set() since that allows us to put in new fields. And we'll pass that detail and $e->getMessage():

55 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 12
class ApiExceptionSubscriber implements EventSubscriberInterface
{
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 17 - 34
                $apiProblem->set('detail', $e->getMessage());
... lines 36 - 45
    }
... lines 47 - 53
}
```

And that should do it, but don't do that...because that will expose the exception message of every exception in our system which is *definitely* not what we want to do.

So there has to be some way for us to determine whether or not it is safe to show the message to the user. There are a number of different ways to do this, FOSRestBundle has some options where you can whitelist on a class by class basis. And that's something we could even do here with an if statement that looks for a set of classes that are safe.

Implementing your own interface is also an option! I'll do something simple here which may or may not work for you, so do think about your project critically when you choose how to do this.

I'll check to see if ($e instanceof HttpExceptionInterface):

55 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 14
15      public function onKernelException(GetResponseForExceptionEvent $event)
16      {
... lines 17 - 33
34              if ($e instanceof HttpExceptionInterface) {
35                  $apiProblem->set('detail', $e->getMessage());
36              }
... lines 37 - 45
46      }
... lines 47 - 55
```

This is used for 404 or 403 errors, so it's typically things that we are in control of.

And you can see here that our 404 error implements that interface which will allow it to be caught by that.

Head back to the terminal and test that guy out:

```
$ ./bin/phpunit -c app --filter test404Exception
```

Beautiful!

Now we have the opportunity to be more helpful to our users whenever we have a 404 error. Or, if you're creating an ApiProblem by hand, you can set the detail field manually.

Chapter 14: Debugging and Cleanup

We're finally to the exciting conclusion, just a few more small cleanup items that we need to take care of.

Starting with debugging. Let's look inside ProgrammerController. What happens if we mess something up? Like some exception gets thrown inside of newAction. To find out let's run just testPOST. As you can see we get a really nice response, but it contains absolutely no details about what went wrong inside of there. That's fine for clients but for debugging it's going to be a nightmare.

If we *are* in debug mode and the status code is 500, I would *love* for Symfony's normal exception handling to take over so we can see that big beautiful stacktrace.

In ApiExceptionSubscriber we'll need to figure out if we're in debug mode. The way to do that is to pass a $debug flag through the __construct method and create a property for it.

I just hit a shortcut called alt+enter. Go to initialize fields, select debug and hit ok. That's just a litte shortcut for PhpStorm to set that flag for me. Before we use that, go into services.yml and pass that value in.

The way to figure out if we're in debug mode is to use %kernel.debug% as an argument.

And if we *are* in debug mode and the status code is 500 we don't want our exception subscriber to do anything. So let's move the status code line up a little bit further, making sure it's after the line where we get the exception. The logic is as simple as if ($statusCode == 500 && $this->debug) then just return. Symfony's normal exception handling will take over from here.

Let's rerun testPOST and it should fail, but I'm hoping I can get some extra details. We get the JSON response still because I changed the request format but we also get the full long stack trace. That is looking really nice, so let's just go ahead and remove the exception.

Tip

There is one thing missing from our listener: logging! In your application, you should inject the logger service and log that an exception occurred. This is important so that you are aware of errors on production. The "finish" download code contains this change.

Thanks to Sylvain for pointing this out in the comments!

type is a URL

Onto the second thing we need to clean up! Inside the spec, under type it says that type should be an absolute URI, and if we put it in our browser, it should take us to the documentation for that. Right now, our types are just strings. We'll fix this in a future episode when we talk properly about documentation, but I at least want to make us kind of follow this rule.

In ApiExceptionSubscriber, instead of calling $apiProblem->toArray(), directly in the JSON response, let's put $data here and create a new $data variable that's set to that. We want to prefix the type key with a URL, except in the case of about:blank - because that's already a URL.

So let's add our if statement, if ($data['type'] != 'about:blank') then,
$data['type'] = 'http://localhost:8000/docs/errors#'.$data['type']; which is just a fake URL for now. But you can get the idea of how we'll eventually put a real URL here to a page where people can look up what those error types actually mean. So that'll be kinda nice.

This stuff may have broken some tests, so let's rerun all of them! Ah yep, and one of them did fail. invalid_body_format failed because we're looking for this exact string and now it's at the end of a URL.

In your test, change assertResponsePropertyEquals to assertResponsePropertyContains which saves me from hardcoding my host name in there:

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

```
... lines 1 - 5
6   class ProgrammerControllerTest extends ApiTestCase
7   {
... lines 8 - 147
148     public function testInvalidJson()
149     {
... lines 150 - 162
163         $this->asserter()->assertResponsePropertyContains($response, 'type', 'invalid_body_format');
164     }
... lines 165 - 175
176 }
```

Copy just that test to our terminal and run it:

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

Perfect, back to green!

Fixing Web Errors

Okay, last thing we need to clean up. This site does have a web interface to it and right now, if I just invent a URL, on the web interface I get a JSON response. This makes sense because the subscriber has completely taken over the error handling for our site, even though, in realitym we only want this to handle errors for our API.

There are a couple of different ways to do this, but at least in our API, everything is under the URL /api. So fixing this is as simple as making our subscriber only do its magic when our URL starts with this. Let's do that!

First get the $request by saying $event->getRequest(). Then let's get our if statement in there. if (strpos()) and we'll look in the haystack which is $request->getPathInfo(), this is the full URL. For the needle use /api and if all of this !== 0, in other words, if the URL doesn't start exactly with /api, then let's just return:

80 lines | src/AppBundle/EventListener/ApiExceptionSubscriber.php

```
... lines 1 - 12
13  class ApiExceptionSubscriber implements EventSubscriberInterface
14  {
... lines 15 - 21
22      public function onKernelException(GetResponseForExceptionEvent $event)
23      {
24          // only reply to /api URLs
25          if (strpos($event->getRequest()->getPathInfo(), '/api') !== 0) {
26              return;
27          }
... lines 28 - 70
71      }
... lines 72 - 78
79  }
```

Head back to the browser and refresh the page. Web interface errors restored!

Let's run the entire test suite to make sure we're done:

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

Look at that, this is a setup to be proud of.

In the next episode we're going to get back to work with pagination, filtering, and other tough but important things with API's.

Alright guys, see ya next time!