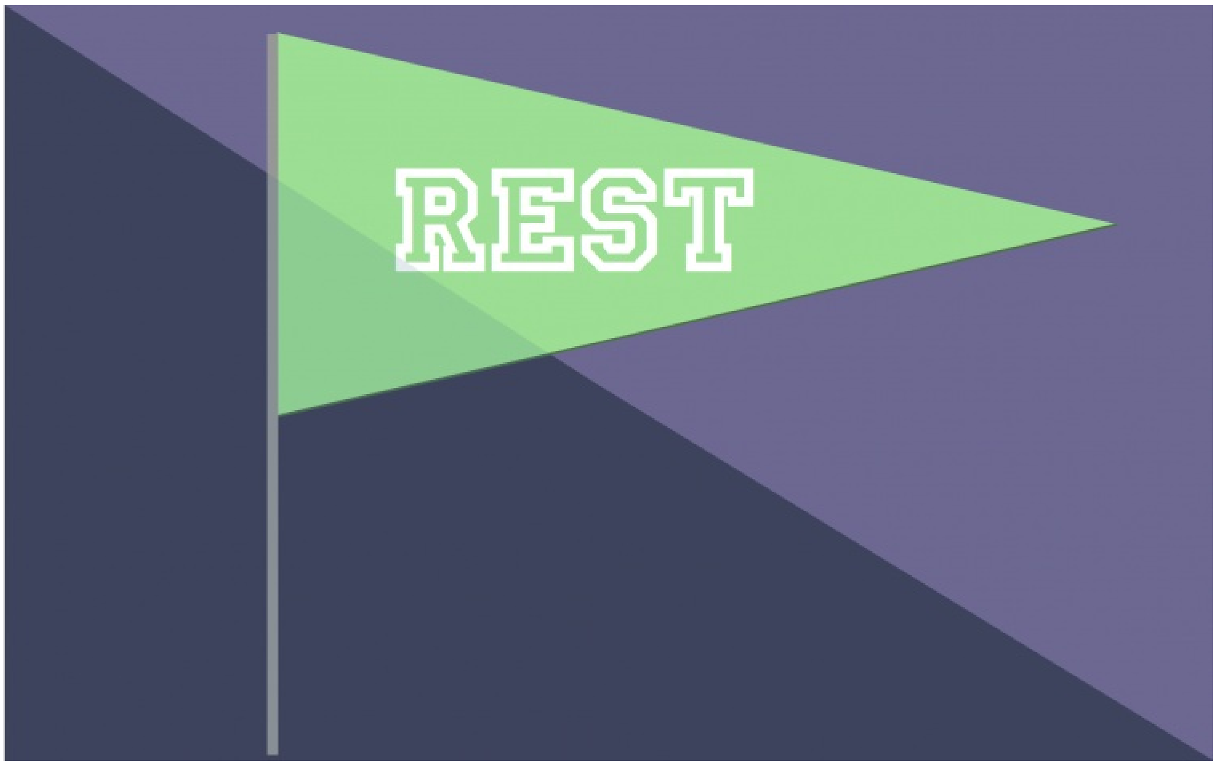
RESTful APIs in the Real World Course 1



With <3 from SymfonyCasts

Chapter 1: The REST API Tutorial

THE REST API TUTORIAL¶

Well hey there! I hope you're ready to work, because we're going to build an API from the ground up! It's not going to be easy: there's a lot to do and a lot of theory to sort through. But I *promise* you that you'll be happy you made the effort.

We'll of course use best practices and learn all about the theory behind REST. But we're also going to be pragmatic. If you stick to the rules too much, you'll get buried in technical draft specifications and blog post. You'll also get the feeling that a perfectly RESTful API might be impossible, and it would probably be pretty tough to use anyways.

Instead, we'll build a really nice API, keep to the best parts of REST, tell you when we're breaking the rules and when the rules are still being debated. And we're not going to stick to the easy stuff. Nope, we'll attack the ugliest areas of an API, like custom methods and where each piece of documentation should live and why.

The Project: Resources and Links¶

The project? Introducing Code Battles: a super-serious site where programmers battle against projects. After you register, you can create a programmer and choose an avatar. REST is based around the idea of resources. If you're using this screencast as a drinking game, you might *not* want to drink each time I say “resource”. It's too important to REST... you won't make it through chapter 2. The same goes for “representations”.

Pay special attention to the links on each page and how resources interact. Our API will feel a lot like the web interface.

With a programmer, you can take an action on her: power up! Based on a little luck, this will increase or decrease the power level of the programmer resource. Next, start a battle, which is between the programmer and a project. A project is our second resource.

My programmer dominated and won the battle! A battle is our third resource, and we can see a collection of them by going to the homepage and clicking “scores”.

Later, I'll explain exactly *why* these are resources, but it should feel natural.

Our world domination plan is to create an API that allows an HTTP client to do everything you just saw, and even more. But what should the endpoint look like for creating a programmer? What about editing it? Should the client send the data via JSON and should we also return it via JSON? How can we communicate validation errors and how should the URL structure look for things like listing programmers and powering up a programmer? What about HTTP methods and status codes? And how can we document all of this? How will the client know what fields to POST when creating a programmer or what URL to use to find projects to battle?

Woh! So building a usable, consistent API involves a lot more than making a few endpoints. But that's why you're here, so let's go!

Chapter 2: HTTP Basics

HTTP BASICS¶

Yep, we need cover a bit of theory. Wait, come back! This stuff is *super* important and *fascinating* too. Put on your thinking cap and let's get to it!

HTTP¶

Everything starts with HTTP: the acronym that describes the format of the request message that a client sends and the response that our server sends back. If you think of an API like a function, the request is the input and the response is our output. It's that simple.

HTTP Request¶

```
GET /api/programmers HTTP/1.1
Host: CodeBattles.io
Accept: application/json,text/html
```

This is a basic request and it has 3 important pieces:

1. `/api/programmers` is the URI: uniform resource identifier. I said **resource**! Each URI is a unique address to a resource, just like you have a unique address to your house. If you have 5 URI's you're saying you have 5 resources.
2. `GET` is the HTTP method and describes what *action* you want to take against the resource. You already know about GET and POST, and possibly also DELETE, PUT and the infamous PATCH. There are others, but mostly we don't care about those.
3. Every line after the first is just a colon-separated list of headers. This request only has two, but a client could send anything.

With that in mind, a POST request might look like this:

```
POST /api/programmers HTTP/1.1
Host: CodeBattles.io
Authorization: Bearer b2gG66D1Tx6d89f3bM6oacHLPSz55j19DEC83x3GkY
Content-Type: application/json

{
  "nickname": "Namespacinator"
}
```

It's the same, except the method is POST and we're sending data in the body of the request. We also have 2 extra headers, one for authentication and one that tells the server that the body has JSON-formatted stuff in it.

HTTP Response¶

The response message is similar:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, private

{
   "nickname": "Namespacinator",
   "avatarNumber": 5,
   "tagLine": "",
   "powerLevel": 25
}
```

The 200 status code is the first important piece, and of course means that everything went just great. Status codes are a *big* part of APIs. But people also like to argue about them. We'll see the important ones as we build.

The headers tell the client that the response is JSON and that the response shouldn't be cached. And of course, the JSON body is at the end.

HTTP is awesome and really simple. Got it? Great, let's move onto something harder.

Chapter 3: REST: Resources and Representations

REST: RESOURCES AND REPRESENTATIONS¶

REST: Representational state transfer. The term was coined famously by Roy Fielding in his doctoral dissertation in 2000. It's complex, and a lot of what makes a REST API hard is understanding and debating the many rules, or constraints laid out in his document.

When you think about an API, it's pretty common to think about its endpoints, in other words the URLs. With REST, if you have a URL, then you have a resource. So, `/programmers/Namespacinator` is probably the address to a single programmer resource and `/programmers` is probably the address to a collection resource of programmers. So even a collection of programmers is considered one resource.

But we already build URLs that work like this on the web, so this is nothing new.

Representations¶

Now that you understand resources, I want to think about representations. Suppose a client makes a GET request to `/programmers/Namespacinator` and gets back this JSON response:

```
{
  "nickname": "Namespacinator",
  "powerLevel": 5
}
```

That's the programmer resource, right? Wrong! No!

This is just a *representation* of the programmer resource. It happens to be in JSON, but the server could have represented the programmer in other ways, like in XML, HTML or even in JSON with a different format.

The same applies when a client sends a request that contains programmer data:

```
POST /api/programmers HTTP/1.1
Host: CodeBattles.io
Authorization: Bearer b2gG66D1Tx6d89f3bM6oacHLPSz55j19DEC83x3GkY
Content-Type: application/json

{
  "nickname": "Namespacinator"
}
```

The client doesn't send a programmer resource, it just sends a representation. The server's job is to interpret this representation and update the resource.

Representation State¶

This is exactly how browsing the web works. An HTML page is *not* a resource, it's just one representation. And when we submit a form, we're just sending a different representation back to the server

One resource could have many representations. Heck, you could get crazy and have an API where you're able to request the XML, JSON *or* HTML representations of any resource. We're just crazy enough that we'll do some of that.

A representation is a machine readable explanation of the current state of a resource.

Yes, I said the current "state" of the resource, and that's another important and confusing term. What REST calls state, you probably think of as simply the "data" of a resource. When the client makes a GET request to `/programmer/Namespacinator`, the JSON is a representation of its current state, or current data. And if the client makes a request to update that programmer, the client is said to be sending a representation in order to update the “state” of the resource.

In REST-speak, a client and server exchange representations of a resource, which reflect its current state or its desired state. REST, or Representational state transfer, is a way for two machines to transfer the state of a resource via representations.

I know I know. We just took an easy idea and made it insane! But if you can understand *this* way of thinking, a lot of what you read about REST will start to make sense.

Chapter 4: Transitions and Client State

TRANSITIONS AND CLIENT STATE¶

Ok, just one more thing: state transitions. We already know about resource state, and how a client can change the resource state by sending a representation with its new state, or data.

In addition to resource state, we also have application state. When you browse the web, you're always on just *one* page of a site. That page is your application's state. When you click a link, you transition your application state to a different page. Easy.

Whatever state we're in, or page we're on, helps us get to the next state or page, by showing us links. A link could be an anchor tag, or an HTML form. If I submit a form it POST's to a URL with some data, and that URL becomes our new app state. If the server redirects us, *that* now becomes our new app state.

Application state is kept in our browser, but the server helps guide us by sending us links, in the form of `a` tags, HTML forms, or even redirect responses. HTML is called a hypermedia format, because it supports having links along with its data.

The same is true for an API, though maybe not the APIs that you're used to. We won't talk about it initially, but a big part of building a RESTful API is sending back links along with your data. These tell you the most likely URLs you'll want your API to follow and are meant to be a guide. When you think about an API client following links, you can start to see how there's application state, even in an API.

Richardson Maturity Model¶

We just accidentally talked through something called the Richardson Maturity Model. It describes different levels of RESTfulness. If your API is built where each resource has a specific URL, you've reached level 1. If you take advantage of HTTP verbs, like GET, POST, PUT and DELETE, congrats: you're level 2! And if you take advantage of these links I've been talking about, that means you've reached "hypermedia", a term we'll discuss later. But anyways, hypermedia means you're a Richardson Maturity Model grand master, or something. How's that for gamification?

We'll keep this model in mind, but for now, let's start building!

Chapter 5: Project Routing

PROJECT ROUTING¶

Ok, let's get start by downloading (see the Download button for subscribers) or cloning the CodeBattles project. Now, follow the README.md file to get things working. It involves downloading Composer and installing the vendor libraries:

```
$ git clone https://github.com/knpuniversity/rest.git
$ cd rest
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar install
```

Tip

If you're new to Composer, watch The Wonderful World of Composer.

When that's done, start up the app by using PHP's awesome built-in web server:

```
$ cd web
$ php -S localhost:8000
```

Note

The built-in web server requires PHP 5.4, which all of you should have! If you're using PHP 5.3, you'll need to configure a VirtualHost of your web server to point at the `web/` directory.

If it worked, then load up the site by going to `http://localhost:8000`. Awesome!

About the App¶

CodeBattles is built in Silex, a PHP microframework. If this is your first time using Silex, take a few minutes with its Documentation to get to know it. It basically let's us design routes, or pages and easily write the code to render those pages. Our setup will look just a little bit different than this, but the idea is the same.

But this is *not* a tutorial on building a REST API on only Silex! Most of what we'll do is basically the same across any framework. You *will* need to do a little bit of work here and there. But trust me, these things are a pleasure to do compared with all the tough REST stuff.

First Endpoint: POST /api/programmers¶

Let's pretend we're building the API for an iPhone app. Ignoring authentication, what's the first thing the user will do in the app? Create a programmer of course! And that's our first API endpoint.

Separate URLs from our Web Interface?¶

But hold up! In the web version of our app, we're already able to create a programmer by filling out a form and submitting it via POST to `/programmers/new`. This either re-renders the HTML page with errors or redirects us.

Why not just reuse the code from this URL and make it work for our API? To do this we'd need to make it accept JSON request data, become smart enough to return errors as JSON and do something other than a redirect on success. Then, `/programmers` could be used by a browser to get HTML *or* by an API client to pass JSON back and forth.

That would be sweet! And later, we'll talk about how you could do that. But for now, things will be a lot easier to understand if we leave the web interface alone, prefix our API URLs with `/api`, and write separate code for it.

This *does* break a rule of REST, because each resource will now have 2 URLs: one for HTML and one for the JSON representation. But REST has a lot of rules, too many for our rebel Codebattles site. We'll break this one, like many APIs. But later, I'll show you how we *could* use 1 URL to return multiple representation.

Basic Routing¶

So let's build the endpoint. Find the `src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php` file and uncomment the route definition:

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

protected function addRoutes(ControllerCollection $controllers)
{
    $controllers->post('/api/programmers', array($this, 'newAction'));
}
```

Next, create a `newAction` inside of this class and return `let's battle!`:

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

public function newAction()
{
    return 'let\'s battle!';
}
```

And just like that, we're making threats and we have a new endpoint with the URL `/api/programmers`. If we make a POST request here, the `newAction` function will be executed and these famous last words will be returned in the response. This is the core of what Silex gives us.

URLs and Resources¶

My choice of a POST request to create a programmer isn't accidental. `/api/programmers` is a collection resource. And according to some HTTP rules I'll show you later, when you want to create a resource, you should send a POST request to its collection.

In other words, I'm not making this all up: I'm following the rules of the web. And in the API world, if you follow the rules, you'll have more friends.

Testing the Endpoint¶

Well let's try it already! That's actually not easy in a browser, since we need to make a POST request. Instead, open up the `testing.php` file at the root of the project that I've already prep'ed for us:

```
// testing.php
require __DIR__.'/vendor/autoload.php';

use Guzzle\Http\Client;

// create our http client (Guzzle)
$client = new Client('http://localhost:8000', array(
    'request.options' => array(
        'exceptions' => false,
    )
));
```

This is a plain PHP file that creates a Guzzle Client object. Guzzle is a simple library for making HTTP requests and receiving responses.

Let's make a POST request to `/api/programmers` and print out the response:

```
// testing.php
// ...
$client = new Client('http://localhost:8000', array(
    'request.options' => array(
        'exceptions' => false,
    )
));

$request = $client->post('/api/programmers');
$response = $request->send();

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

Try it out by running the file from the command line. You’ll need to open a new terminal tab and make sure you’re at the root of the project where the file is:

```
$ php testing.php
```

```
HTTP/1.1 200 OK
Host: localhost:8000
Connection: close
Cache-Control: no-cache
Content-Type: text/html; charset=UTF-8

let's battle!
```

Success!

Chapter 6: POST: Creation, Location Header and 201

POST: CREATION, LOCATION HEADER AND 201¶

Once the POST endpoint works, the client will send programmer details to the server. In REST-speak, it will send a representation of a programmer, which can be done in a bunch of different ways. It's invisible to us, but HTML forms do this by sending data in a format called `application/x-www-form-urlencoded`:

```
POST /api/programmers HTTP/1.1
Host: localhost:8000
Content-Type: application/x-www-form-urlencoded

nickname=Geek+Dev1&avatarNumber=5
```

PHP reads this and puts it into the `$_POST` array. That's ok for the web, but in the API world, it's ugly. Why not, have the client send us the representation in a beautiful bouquet of curly braces known as JSON:

```
POST /api/programmers HTTP/1.1
Host: localhost:8000
Content-Type: application/json

{
    "nickname": "Geek Dev1",
    "avatarNumber": 5
}
```

Creating a request like this with Guzzle is easy:

```php
// testing.php
// ...

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

$request = $client->post('/api/programmers', null, json_encode($data));
$response = $request->send();

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

The second `null` argument is the request headers to send. We're not worried about headers yet, so we can just leave it blank.

Coding up the Endpoint¶

Back in the `ProgrammerController` class, let's make this work by doing our favorite thing - coding! First, how do we get the JSON string passed in the request? In Silex, you do this by getting the `Request` object and calling `getContent()` on it. Let's just return the data from the endpoint so we can see it:

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

public function newAction(Request $request)
{
    $data = $request->getContent();
    return $data;
}
```

Tip

Your framework will likely have a shortcut for getting the request content or body. But if it doesn't, you can get it by using this strange bit of code:

```
$data = file_get_contents('php://input');
```

This is a special stream that reads the request body. For more details, see php.net: php://.

Try running our `testing.php` file again:

```
$ php testing.php
```

This time, we see the JSON string being echo'ed right back at us:

```
HTTP/1.1 200 OK
...
Content-Type: text/html; charset=UTF-8

{"nickname":"ObjectOrienter31","avatarNumber":5}
```

Creating the Programmer Resource Object¶

Awesome! I've already created a `Programmer` class, which has just a few properties on it. I also created simple classes for the other two resources - `Project` and `Battle`. We'll use these later.

In `newAction`, we have the JSON string, so let's decode it and use the data to create a new `Programmer` object that's ready for battle. We'll use each key that the client sends to populate a property on the object:

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

public function newAction(Request $request)
{
    $data = json_decode($request->getContent(), true);

    $programmer = new Programmer($data['nickname'], $data['avatarNumber']);
    $programmer->tagLine = $data['tagLine'];
    // ...
}
```

Note

Do not forget to add `use KnpU\CodeBattle\Model\Programmer;` namespace for the `Programmer` class at the beginning of the `ProgrammerController.php`.

My app also has a really simple ORM that lets us save these objects to the database. How you save things to your database will be different. The key point is that we have a `Programmer` class that models how we want our API to look, and that we can somehow save this:

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

public function newAction(Request $request)
{
    $data = json_decode($request->getContent(), true);

    $programmer = new Programmer($data['nickname'], $data['avatarNumber']);
    $programmer->tagLine = $data['tagLine'];
    $programmer->userId = $this->findUserByUsername('weaverryan')->id;

    $this->save($programmer);

    return 'It worked. Believe me - I\'m an API';
}
```

At the bottom, I’m just returning a really reassuring message that everything went ok.

Faking the Authenticated User¶

I’ve also added one really ugly detail:

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

Every programmer is created and owned by one user. On the web, finding out *who* is creating the programmer is as easy as finding out which user is currently logged in.

But our API has no idea who *we* are - we’re just a client making requests without any identification.

We’ll fix this later. Right now, I’ll just make *every* programmer owned by me. Make sure to use my username: it’s setup as test data that’s always in our database. This test data is also known as fixtures.

Ok, the moment of truth! Run the testing script again:

```
$ php testing.php
```

```
HTTP/1.1 200 OK
Host: localhost:8000
...
Content-Type: text/html; charset=UTF-8

It worked. Believe me - I'm an API
```

The message tells us that it probably worked. And if you login as `weaverryan` with password `foo` on the web, you’ll see this fierce programmer-warrior in the list.

Status Code 201¶

But no time to celebrate! Our response is a little sad. First, since we’ve just created a resource, the HTTP elders say that we should return a 201 status code. In Silex, we just need to return a new `Response` object and set the status code as the second argument:

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

public function newAction(Request $request)
{
    // ...
    $this->save($programmer);

    return new Response('It worked. Believe me - I\'m an API', 201);
}
```

Running the testing script this time shows us a 201 status code.

Location Header¶

And when we use the 201 status code, there’s another rule: include a `Location` header that points to the new resource. Hmm, we don’t have an endpoint to get a single programmer representation yet. To avoid angering the RESTful elders, let’s add a location header, and just fake the URL for now:

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

public function newAction(Request $request)
{
    // ...
    $this->save($programmer);

    $response = new Response('It worked. Believe me - I\'m an API', 201);
    $response->headers->set('Location', '/some/programmer/url');

    return $response;
}
```

If you stop and think about it, this is how the web works. When we submit a form to create a programmer, the server returns a redirect that takes us to the page to view it. In an API, the status code is 201 instead of 301 or 302, but the server is trying to help show us the way in both cases.

Try the final product out in our test script:

```
$ php testing.php
```

```
HTTP/1.1 201 Created
...
Location: /some/programmer/url
Content-Type: text/html; charset=UTF-8

It worked. Believe me - I'm an API
```

Other than the random text we’re still returning, this endpoint is looking great. Now to GET a programmer!

Chapter 7: GET’ing Resources, and Content-Type

GET’ING RESOURCES, AND CONTENT-TYPE¶

We just created ObjectOrienter – how tough and scary– now let’s make an endpoint that’s able to fetch that programmer representation. Start by writing how it’ll look when a client makes the request:

```
// testing.php
// ...

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

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

The URL is `/api/programmers/{nickname}`, where the `nickname` part changes based on which programmer you want to get. In a RESTful API, the URL structures don’t actually matter. But to keep your sanity, if `/api/programmers` returns the collection of programmers, then make `/api/programmers/{id}` return a single programmer, where `{id}` is something unique. And be consistent: if your collection resources are plural - like `/api/programmers`, use the plural form for all collection resources. If you make these URLs inconsistent you are going to make your future self really really miserable. Be consistent and your API users will leave you lots of happy emoticons.

Basic Routing and Controller¶

To make this endpoint work, go back to `ProgrammerController`, and add another routing line. But this time, make the URL respond to the `GET` method:

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

protected function addRoutes(ControllerCollection $controllers)
{
    $controllers->post('/api/programmers', array($this, 'newAction'));

    $controllers->get('/api/programmers/{nickname}', array($this, 'showAction'));
}
```

The `{nickname}` in the URL means that this route will be matched by any GET request that looks like `/api/programmers/*`. Next, make a `showAction` method and just return a simple message:

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

public function showAction($nickname)
{
    return 'Hello '.$nickname;
}
```

If we go to `/api/programmers/foo`, the `$nickname` variable will be equal to `foo`. This is special to Silex, but you can do this kind of stuff with any framework.

Ok, try out the testing script:

```
$ php testing.php
```

```
HTTP/1.1 200 OK
# ...
Content-Type: text/html; charset=UTF-8

Hello ObjectOrienter366
```

Returning a JSON Response¶

Our goal is to return a representation of the programmer resource. This could be in JSON, XML and if it’s April Fools try some invented format. We’ll use JSON, because it’s easy simple, and all languages support it. To do this, first query for the programmer by using its nickname. I’ll do this using a `findOneByNickname` method from my simple ORM:

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

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

    // ...
}
```

This returns a `Programmer` object. The code you use to query for data will be different. The really important part is to finish with an object that has all the data you want to send back in your API.

Next, just turn the `Programmer` object into an array manually. And finally, create a new `Response` object just like the POST endpoint. But this time, set its body to be the JSON-encoded string:

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

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

    $data = array(
        'nickname' => $programmer->nickname,
        'avatarNumber' => $programmer->avatarNumber,
        'powerLevel' => $programmer->powerLevel,
        'tagLine' => $programmer->tagLine,
    );

    return new Response(json_encode($data), 200);
}
```

The correct status code is 200. We’ll learn about other status codes, but you’ll still use the good ol’ 200 in most cases, especially for GET requests.

Test it out!

```
$ php testing.php
```

```
HTTP/1.1 200 OK
# ...
Content-Type: text/html; charset=UTF-8

{"nickname":"ObjectOrienter135","avatarNumber":"5","powerLevel":"0","tagLine":"a test dev!"}
```

But what's the Content-Type?¶

Perfect! Except that we're still telling the client that the content is written in HTML. That's the job of the `Content-Type` response header, and it defaults to `text/html`. Our response is being dishonest right now, and we risk confusing an API client.

Fix this by manually setting the `Content-Type` header on the `Response` before returning it:

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

public function showAction($nickname)
{
    // ...

    $response = new Response(json_encode($data), 200);
    $response->headers->set('Content-Type', 'application/json');

    return $response;
}
```

How you set a response header may be different in your app, but there is definitely a way to do this. And because the `Content-Type` header is so important, you may even have a shortcut method for it.

Note

For example, in Laravel, you can return a JSON response (with correct `Content-Type`) with:

```
Response::json(array('name' => 'Steve', 'state' => 'CA'));
```

We're now returning a JSON representation of the resource, setting its `Content-Type` header correctly and using the right status code. Great work.

404 Pages¶

But let's not forget to return a 404 if we're passed a bad nickname. In our app, I've created a shortcut for this called `throw404`:

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

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

    if (!$programmer) {
        $this->throw404('Crap! This programmer has deserted! We\'ll send a search party');
    }

    // ...
}
```

Under the surface, this throws a special type of exception that's converted by Silex into a 404 response. In your app, just return a 404 page however you normally do.

Try it out by temporarily changing our testing script to point to a made-up nickname:

```php
// testing.php
// ...

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

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

When we run the script now, we *do* see a 404 page, though it's a big ugly HTML page instead of JSON. We'll talk about properly handling API errors later.

Chapter 8: Updating the Location Header

UPDATING THE LOCATION HEADER¶

Hey, we have a working endpoint to view a single programmer! We're awesome :) Now do you remember the `Location` response header we return after creating a new programmer? Let's update that to be a real value.

To do this, first add a `bind` function to our programmer route:

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

protected function addRoutes(ControllerCollection $controllers)
{
    $controllers->post('/api/programmers', array($this, 'newAction'));

    $controllers->get('/api/programmers/{nickname}', array($this, 'showAction'))
        ->bind('api_programmers_show');
}
```

This gives the route an internal name of `api_programmers_show`. We can use that below to generate a proper URL to the new programmer resource:

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

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

    $response = new Response('It worked. Believe me - I\'m an API', 201);
    $programmerUrl = $this->generateUrl(
        'api_programmers_show',
        ['nickname' => $programmer->nickname]
    );
    $response->headers->set('Location', $programmerUrl);

    return $response;
}
```

The `generateUrl` method is a shortcut I added for our app, and it combines the `nickname` with the rest of the URL. You may make URLs differently in your app, but the idea is the same: set the `Location` header to the URI where I can GET this new resource.

Tip

The `generateUrl` method is just a shortcut for doing this:

```
$programmerUrl = $this->container['url_generator']->generate(
    'api_programmers_show',
    ['nickname' => $programmer->nickname]
);
```

Update the `testing.php` script to print out the response from the original POST so we can check this out:

```
// testing.php
// ...

// 1) Create a programmer resource
$request = $client->post('/api/programmers', null, json_encode($data));
$response = $request->send();

echo $response;
echo "\n\n";
die;

// 2) GET a programmer resource
// ...
```

And when we run it again, we've got a working `Location` header:

```
HTTP/1.1 201 Created
...
Location: /api/programmers/ObjectOrienter330

It worked. Believe me - I'm an API
```

Using the Location Header¶

The `Location` header is more than just a nice thing. Its purpose is to help the client know where to go next without needing to hardcode URLs or URL patterns. To prove this, we can update our testing script to read the `Location` header and use it for the next request. This lets us *remove* the hardcoded URL pattern we had before:

```
// testing.php
// ...

// 1) Create a programmer resource
$request = $client->post('/api/programmers', null, json_encode($data));
$response = $request->send();

$programmerUrl = $response->getHeader('Location');

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

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

If the URL pattern to view a programmer changes in the future, our client won't break. That's really powerful. But it's also where things start to get complicated. More on that later, dear warrior.

Chapter 9: GET /programmers: A collection of Programmers

GET /PROGRAMMERS: A COLLECTION OF PROGRAMMERS¶

We now have 2 URLs and so 2 resources:

- `/api/programmers`, which represents a collection of resources (i.e. all programmers);
- `/api/programmers/{nickname}`, which represents one programmer.

Actually, we can POST to the `/api/programmers` resource, but we can't GET it yet. And nothing says that we *have* to support the GET method for a resource. But we'll add it for two reasons. First, I'll pretend that our imaginary iPhone app needs it. And second, API users tend to assume that you can GET most any resource. If we make this possible, our API is that much more predictable and friendly.

Coding up our Test¶

Like always, let's start by updating our testing script to try the new endpoint:

```
// testing.php
// ...

// 3) GET a list of all programmers
$request = $client->get('/api/programmers');
$response = $request->send();

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

Creating the Route¶

Next, create a new route that points to a new `listAction` method in our `ProgrammerController` class:

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

protected function addRoutes(ControllerCollection $controllers)
{
    // the 2 other routes ...

    $controllers->get('/api/programmers', array($this, 'listAction'));
}
```

Creating the Endpoint¶

I'll copy the `showAction` and modify it for `listAction`. First, remove the `$nickname` argument, since there's no `{nickname}` in this new URI. Next, query for *all* programmers using another method from my ORM. The key is that this gives us an array of `Programmer` objects:

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

public function listAction()
{
    $programmers = $this->getProgrammerRepository()->findAll();
    // ...
}
```

Like before, we need to turn each Programmer into an array. I really don't want to duplicate this logic. Instead, create a new `` `serializeProgrammer `` function right in this class that converts a `Programmer` object into an array. This is still manual, but later I'll show you some fancier ways to do this:

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

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

Now, use this to build a big array of the programmers in `listAction`:

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

public function listAction()
{
    $programmers = $this->getProgrammerRepository()->findAll();
    $data = array('programmers' => array());
    foreach ($programmers as $programmer) {
        $data['programmers'][] = $this->serializeProgrammer($programmer);
    }

    $response = new Response(json_encode($data), 200);
    $response->headers->set('Content-Type', 'application/json');

    return $response;
}
```

And make sure to also update `showAction` to use `serializeProgrammer`:

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

public function showAction($nickname)
{
    // ...

    // replace the manual creation of the array with this function call
    $data = $this->serializeProgrammer($programmer);

    // ...
}
```

Cool - let's try it!

```
$ php testing.php
```

```
HTTP/1.1 200 OK
...
Content-Type: application/json

{
   "programmers": [
      {
         "nickname":"ObjectOrienter14",
         "avatarNumber":"5",
         "powerLevel":"0",
         "tagLine":null
      },
      {
         "nickname":"ObjectOrienter795",
         "avatarNumber":"5",
         "powerLevel":"0",
         "tagLine":"a test dev!"
      }
   ]
}
```

What JSON Structure to Use?¶

Awesome! So why did I put the data under a `programmers` key? Actually, no special reason, I just invented this standard. I could have structured my JSON however I wanted.

And actually, there are some pre-existing standards that exist on the web for organizing your JSON structures. These answer questions like, “should I put the data under a `programmers` key?” or “how should I organize details on how to paginate through the results?”.

This is real important stuff, and we’ll go into more detail later. For now, we just have to follow one golden rule: find a standard and be consistent.

Now rewind back 10 seconds and listen to that again at least 5 times. Consistency people!

Chapter 10: Fixing the Content-Type on POST

FIXING THE CONTENT-TYPE ON POST¶

Check us out! We now have 3 working endpoints, but one has a big issue. The POST *still* returns a text string as its response. Even if you don't know what it *should* return, that's embarassing. Come on, we can do better!

After creating a resource, one great option is to return a representation of the new resource. Use the `serializeProgrammer` to get JSON and put it into the Response:

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

public function newAction(Request $request)
{
    // ...
    $this->save($programmer);

    $data = $this->serializeProgrammer($programmer);
    $response = new Response(
        json_encode($data),
        201
    );
    $programmerUrl = $this->generateUrl(
        'api_programmers_show',
        ['nickname' => $programmer->nickname]
    );
    $response->headers->set('Location', $programmerUrl);
    $response->headers->set('Content-Type', 'application/json');

    return $response;
}
```

If the client needs that information, then we've just saved them one API request.

And don't forget to set the `Content-Type` response header to `application/json`. To see your handy work, print out that response temporarily and try it:

```
// testing.php
// ...

// 1) Create a programmer resource
$request = $client->post('/api/programmers', null, json_encode($data));
$response = $request->send();

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

A JsonResponse Shortcut¶

And actually, since returning JSON is so common, Silex has a shortcut: the `JsonResponse` class. It takes care of running `json_encode` *and* setting the `Content-Type` header for us – double threat!:

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

public function newAction(Request $request)
{
    // ...
    $this->save($programmer);

    $data = $this->serializeProgrammer($programmer);
    $response = new JsonResponse($data, 201);
    $programmerUrl = $this->generateUrl(
        'api_programmers_show',
        ['nickname' => $programmer->nickname]
    );
    $response->headers->set('Location', $programmerUrl);

    return $response;
}
```

That's just there for convenience, but it cuts down on some code. If your framework or application doesn't have anything like this, create a class or function to help with this: it will go a long way towards following our favorite motto: be consistent.

Finding Spec Information¶

By the way, how do I know these rules, like that a 201 response should have a `Location` header or that it should return the entity body? These guidelines come from the IETF and the W3C in the form of big technical documents called RFC's. They're not always easy to interpret, but sometimes they're awesome. For example, if you google for `http status 201` you'll find the famous RFC 2616, which gives us the details about the 201 status code and most of the underlying guidelines for how HTTP works.

I'll help you navigate these rules. But as we go, try googling for answers and seeing what's out there. Some RFC's, like 2616, are older and well adopted. Others are still up for comment and being interpreted. Some of which we'll cover later.

Chapter 11: Testing your API with PHPUnit

TESTING YOUR API WITH PHPUNIT¶

What, a testing chapter so early? Yep, with API's, you really can't avoid it, and that's a good thing. One way or another, to develop, test and debug each endpoint, you'll need to either write some code or configure an HTTP browser plugin like Postman. And if you're already doing all that work, you might as well make these calls repeatable and automated.

Guzzle and PHPUnit¶

Like I mentioned, if you're making HTTP requests in PHP, you'll want to use Guzzle, like we are in our `testing.php` file. So, the easiest way to create tests is just to put PHPUnit and Guzzle together.

I already have PHPUnit installed in our app via Composer, so let's go straight to writing a test. Create a new `Tests` directory and put a `ProgrammerControllerTest.php` file there. Create a class inside and extend the normal PHPUnit base class:

```php
// src/KnpU/CodeBattle/Tests/ProgrammerControllerTest.php
namespace KnpU\CodeBattle\Tests;

class ProgrammerControllerTest extends \PHPUnit_Framework_TestCase
{

}
```

Next, add a `testPOST` method and copy in the POST logic from the `testing.php` script:

```php
// src/KnpU/CodeBattle/Tests/ProgrammerControllerTest.php
// ...

public function testPOST()
{
    // create our http client (Guzzle)
    $client = new Client('http://localhost:8000', array(
        'request.options' => array(
            'exceptions' => false,
        )
    ));

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

    $request = $client->post('/api/programmers', null, json_encode($data));
    $response = $request->send();
}
```

Finally, let's add some asserts to check that the status code is 201, that we have a `Location` header and that we get back valid JSON:

```
// src/KnpU/CodeBattle/Tests/ProgrammerControllerTest.php
// ...

public function testPOST()
{
    // ...

    $request = $client->post('/api/programmers', null, json_encode($data));
    $response = $request->send();

    $this->assertEquals(201, $response->getStatusCode());
    $this->assertTrue($response->hasHeader('Location'));
    $data = json_decode($response->getBody(true), true);
    $this->assertArrayHasKey('nickname', $data);
}
```

To try it out, use the `vendor/bin/phpunit` executable and point it at the test file.

```
$ php vendor/bin/phpunit src/KnpU/CodeBattle/Tests/ProgrammerControllerTest.php
```

With any luck, Sebastian Bergmann will tell you that everything is ok! Of course I never trust a test that passes on the first try, so be sure to change things and make sure it fails when it should too.

Chapter 12: Behat for Testing

BEHAT FOR TESTING¶

The great thing about using PHPUnit is that it’s dead-simple: make an HTTP request and assert some things about its response. If you want to test your APIs using Guzzle and PHPUnit, you’ll be very successful and your office will smell of rich mahogany.

But in our app, we’re going to make our tests much more interesting by using a tool called Behat. If you’re new to Behat, you’re in for a treat! But also don’t worry: we’re going to use Behat, but not dive into it too deeply. And when you want to know more, watch our Behat Screencast and then use the code that comes with this project to jumpstart testing your API.

Creating Scenarios¶

With Behat, we write human-readable statements, called scenarios, and run these as tests. To see what I mean, find the `features/api/programmer.feature` file:

```
# api/features/programmer.feature
Feature: Programmer
  In order to battle projects
  As an API client
  I need to be able to create programmers and power them up

  Background:
    # Given the user "weaverryan" exists

  Scenario: Create a programmer
```

As you’ll see, each feature file will contain many scenarios. I’ll fill you in with more details as we go. For now, let’s add our first scenario: *`Create a Programmer`*:

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

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

I’m basically writing a user story, where our user is an API client. This describes a client that makes a POST request with a JSON body. It then checks to make sure the status code is 201, that we have a `Location` header and that the response has a `nickname` property.

Running Behat¶

I may sound crazy, but let’s execute these english sentences as a real test. To do that, just run the `behat` executable, which is in the `vendor/bin` directory:

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

Green colors! It says that 1 scenario passed. In the background, a real HTTP request was made to the server and a real response was sent back and then checked. In our browser, we can actually see the new `ObjectOrienter` programmer.

Configuring Behat¶

Oh, and it knows what our hostname is because of a config file: `behat.yml.dist`. We just say `POST /api/programmers` and it knows to make the HTTP request to `http://localhost:8000/api/programmers`.

Note

If you're running your site somewhere other than `localhost:8000`, copy `behat.yml.dist` to `behat.yml` and modify the `base_url` in both places.

How Behat Works¶

Behat looks like magic, but it's actually really simple. Open up the `ApiFeatureContext` file that lives in the `features/api` directory. If we scroll down, you'll immediately see functions with regular expressions above them:

```
// features/api/ApiFeatureContext.php
// ...

/**
 * @When /^I request "(GET|PUT|POST|DELETE|PATCH) ([^"]*)"$/
 */
public function iRequest($httpMethod, $resource)
{
    // ...
}
```

Behat reads each line under a scenario and then looks for a function here whose regular expression matches it. So when we say `I request "POST /api/programmers"`, it calls the `iRequest` function and passes `POST` and `/api/programmers` as arguments. In there, our old friend Guzzle is used to make HTTP requests, just like we're doing in our `testing.php` script.

Note

Hat-tip to Phil Sturgeon and Ben Corlett who originally created this file for Phil's Build APIs you Won't Hate book.

Also, a KnpU (Johan de Jager) user has ported the ApiFeatureContext to work with Guzzle 6 and Behat 3. You can find it here: https://github.com/thejager/behat-api-feature-context.

To sum it up: we write human readable sentences, Behat executes a function for each line and those functions use Guzzle to make real HTTP requests. Behat is totally kicking butt for us!

Seeing our Library of Behat Sentences¶

I created this file and filled in all of the logic in these functions. This gives us a big library of language we can use immediately. To see it, run the same command with a `-dl` option:

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

Anywhere you see the quote-parentheses mess that's a wildcard that matches anything. So as long as we write scenarios using this language, we can test without writing any PHP code in `ApiFeatureContext`. That's powerful.

If you type a line that doesn't match, Behat will print out a new function with a new regular expression. It's Behat's way of saying "hey, I don't have that language. So if you want it, paste this function into ApiFeatureContext and fill in the guts yourself". I've already prepped everything we need. So if you see this, you messed up - check your spelling!

And if using Behat is too much for you right now, just keep using the PHPUnit tests with Guzzle, or even use a mixture!

Chapter 13: Handling Data in Tests

HANDLING DATA IN TESTS¶

Let's run our test a second time:

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

It fails! The nickname of a programmer is unique in the database, and if you look closely, this fails because the API tries to insert another `ObjectOrienter` and blows up. To fix this, add a new function in `ApiFeatureContext` with a special `@BeforeScenario` anotation above it:

```
// features/api/ApiFeatureContext.php
// ...

/**
 * @BeforeScenario
 */
public function clearData()
{
    $this->getProjectHelper()->reloadDatabase();
}
```

The body of this function is specific to my app - it calls out to some code that truncates all of my tables. If you can write code to empty your database tables, or at least the ones we'll be messing with in our tests, then you can do this.

Tip

In order to access your framework's normal database-related functions, you'll need to bootstrap your app inside this class. For many frameworks, libraries exist to glue Behat and it together. If you have issues or questions, feel free to post them in the comments.

The `@BeforeScenario` annotation, or comment, tells Behat to automatically run this before every scenario. This guarantees that we're starting with a very predictable, empty database before each test.

Using Background to Add a User¶

Try the test again:

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

Dang, it failed again. Ah, remember how we're relating all programmers to the `weaverryan` user? Well, when we empty the tables before the scenario, this user gets deleted too. That's expected, and I already have a sentence to take care of this. Uncomment the `Background` line above the scenario. This runs a function that inserts my user:

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

  Background:
    Given the user "weaverryan" exists

  # ...
```

Eventually we'll have many scenarios in this one file. Lines below `Background` are executed before each `Scenario`. Ok, try it one more time!

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

Success! When you test, it’s critical to make sure that your database is in a predictable state before each test. Don’t assume that a user exists in your database: create it with a scenario or background step.

And, every test, or scenario in Behat, should work independently. So don’t make one scenario depend on the data of a scenario that comes before it. That’s a huge and common mistake. Eventually, it’ll make your tests unpredictable and hard to debug. If you do a little bit of work early on to get all this data stuff right, you and Behat are going to be very happy together.

Test: GET One Programmer¶

Let’s add a second scenario for making a GET request to view a single programmer. This entirely uses language that I’ve already prepped for us:

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

The `Given` statement actually inserts the user into the database before we start the test. That’s exactly what I was just talking about: if I need a user, write a scenario step that adds one.

The rest of the test just checks the status code and whatever data we think is important, just like in the previous scenario.

Run it!

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

Success!

Test: GET all Programmers¶

We’re on a roll at this point, so let’s add a third scenario for making a GET request to see the collection of all programmers. Oh, and the title that we give to each scenario - like `GET one programmer` : is just for our benefit, it’s not read by Behat. And for that matter, neither are the first 4 lines of the feature file. But you should still learn more about the importance of these - don’t skip them!

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

Scenario: GET a collection of programmers
  Given the following programmers exist:
    | nickname    | avatarNumber |
    | UnitTester  | 3            |
    | CowboyCoder | 5            |
  When I request "GET /api/programmers"
  Then the response status code should be 200
  And the "programmers" property should be an array
  And the "programmers" property should contain 2 items
```

Here, we insert 2 programmers into the database before the test, make the HTTP request and then check some basic things on the response. It’s the same, boring process over and over again.

I hope you’re seeing how awesome testing our API with Behat is going to be!

Chapter 14: PUT: Editing Resources

PUT: EDITING RESOURCES¶

We can create a programmer resource, view a representation of a programmer, or view the collection representation for all programmers.

PUT: The Basic Definition¶

We're killing it! Now let's make it possible to edit a programmer resource.

Depending on who you ask, there are about 10 HTTP methods, and the 4 main ones are

- GET
- POST
- PUT
- DELETE

We know GET is for retrieving a representation and DELETE is pretty clear.

But things get trickier with POST and PUT. I'm about to say something that's **incorrect**. Ready?

POST is used for creating resources and PUT is used for updating.

Seriously, this is **not true**, and it's dangerous to say: there might be hardcore REST fans waiting around any corner that's eager to tell us how wrong that is.

But in practice, this statement is pretty close: PUT is for edit, POST is for create. So let's use the PUT method for our edit endpoint. Afterwards, we'll geek out on the *real* difference between POST and PUT.

Writing the Test¶

You guys know the drill: we start by writing the test. So let's add yet *another* scenario:

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

Scenario: PUT to update a programmer
  Given the following programmers exist:
    | nickname    | avatarNumber | tagLine |
    | CowboyCoder | 5            | foo     |
  And I have the payload:
    """
    {
      "nickname": "CowboyCoder",
      "avatarNumber" : 2,
      "tagLine": "foo"
    }
    """
  When I request "PUT /api/programmers/CowboyCoder"
  Then the response status code should be 200
  And the "avatarNumber" property should equal "2"
```

This looks a lot like our POST scenario, and that's good: consistency! But we *do* need to add a line to put a programmer into the database and tweak a few other details. The status code *is* different: 201 is used when an asset is created but the normal 200 is used when it's an update.

Just to keep us tied into the theory of things, I'll describe this using REST-nerd language. Ready? Ok.

This tests that when we send a "representation" of a programmer resource via PUT, the server will use it to update that resource and return a representation.

We haven't actually coded this yet, so when we run the test, it fails:

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

The test reports that the status code isn't 200, it's 405. 405 means "method not allowed", and our framework is doing this for us. It's a way of saying "Hey, `/api/programmers/CowboyCoder` *is* a valid URI, but the resource doesn't support the PUT method."

If your API doesn't support an HTTP method for a resource, you should return a 405 response. If you use a decent framework, you should get this functionality for free.

Coding up the PUT Endpoint¶

Let's code it! Create another route, but use the `put` method to make it respond only to PUT requests:

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

protected function addRoutes(ControllerCollection $controllers)
{
    // ...

    $controllers->put('/api/programmers/{nickname}', array($this, 'updateAction'));
}
```

Next, copy the `newAction` and rename it to `updateAction`, because these will do almost the same thing. The biggest difference is that instead of creating a new `Programmer` object, we'll query the database for an existing object and update it. Heck, we can steal that code from `showAction`:

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

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

    if (!$programmer) {
        $this->throw404();
    }

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

    $programmer->nickname = $data['nickname'];
    $programmer->avatarNumber = $data['avatarNumber'];
    $programmer->tagLine = $data['tagLine'];
    $programmer->userId = $this->findUserByUsername('weaverryan')->id;

    $this->save($programmer);

    // ...
}
```

Change the status code from 201 to 200, since we're no longer creating a resource. And you should also remove the `Location` header:

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

public function updateAction($nickname, Request $request)
{
    // ...

    $this->save($programmer);

    $data = $this->serializeProgrammer($programmer);

    $response = new JsonResponse($data, 200);

    return $response;
}
```

We only need this header with the 201 status code when a resource is created. And it makes sense: when we create a new resource, we don't know what its new URI is. But when we're editing an existing resource, we clearly already have that URI, because we're using it to make the edit.

Run the Test and Celebrate¶

Time to run the test!

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

Woot! It passes! And we can even run it over and over again.

Chapter 15: Debugging Tests

DEBUGGING TESTS¶

But what if this *had* failed? Could we debug it?

Let’s pretend we coded something wrong by throwing a big ugly exception in our controller:

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

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

    if (!$programmer) {
        $this->throw404();
    }

    throw new \Exception('This is scary!');

    // ...
}
```

Now run the test again:

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

Not surprisingly, we’re getting a 500 error instead of 200. But we can’t really see what’s going on because we can’t see the big error page!

But don’t worry! First, I’ve done my best to configure Behat so when something fails, part of the last response is printed below.

Tip

This functionality works by returning the h1 and h2 elements of the HTML page. If your app shows errors with different markup, tweak the `ApiFeatureContext::printLastResponseOnError` method to your liking.

If this doesn’t tell you enough, we can print out the last response in its entirety. To do this, add “And print last response” to our scenario, just *before* the failing line:

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

Scenario: PUT to update a programmer
  Given the following programmers exist:
    | nickname    | avatarNumber | tagLine |
    | CowboyCoder | 5            | foo     |
  And I have the payload:
    """
    {
      "nickname": "CowboyCoder",
      "avatarNumber" : 2,
      "tagLine": "foo"
    }
    """
  When I request "PUT /api/programmers/CowboyCoder"
  And print last response
  Then the response status code should be 200
  And the "avatarNumber" property should equal "2"
```

Now just re-run the test:

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

It may be ugly, but the entire response of the last request our test made is printed out, including all the header information on top. Once you've figured out and fixed the problem, just take the `print last response` line out and keep going!

Chapter 16: PUT: Killing Duplicated Code

PUT: KILLING DUPLICATED CODE¶

Our tests are passing, but we're doing a bad job: I've created duplicated code in `newAction` and `updateAction`!

Let's redeem ourselves! Create a new private function called `handleRequest` and copy the code into it that reads the request body and sets the data on the Programmer:

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

private function handleRequest(Request $request, Programmer $programmer)
{
    $data = json_decode($request->getContent(), true);

    $programmer->nickname = $data['nickname'];
    $programmer->avatarNumber = $data['avatarNumber'];
    $programmer->tagLine = $data['tagLine'];
    $programmer->userId = $this->findUserByUsername('weaverryan')->id;
}
```

Cool! Now we can just call this from `newAction` and `updateAction`:

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

public function newAction(Request $request)
{
    $programmer = new Programmer();
    $this->handleRequest($request, $programmer);
    $this->save($programmer);

    // ...
}

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

    if (!$programmer) {
        $this->throw404();
    }

    $this->handleRequest($request, $programmer);
    $this->save($programmer);

    // ...
}
```

Re-run the tests to see if we broke anything:

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

Cool! I'm going to change how this code is written *just* a little bit so that it's even more dynamic:

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

private function handleRequest(Request $request, Programmer $programmer)
{
    $data = json_decode($request->getContent(), true);

    if ($data === null) {
        throw new \Exception(sprintf('Invalid JSON: '.$request->getContent()));
    }

    // determine which properties should be changeable on this request
    $apiProperties = array('nickname', 'avatarNumber', 'tagLine');

    // update the properties
    foreach ($apiProperties as $property) {
        $val = isset($data[$property]) ? $data[$property] : null;
        $programmer->$property = $val;
    }

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

There's nothing important in this change, but it'll make some future changes easier to understand. If you're using a form library or have a fancier ORM, you can probably do something like this with even less code than I have.

While we're here, let's throw a big exception if the client sends us invalid JSON.

Chapter 17: GET Representation != POST Representation

GET REPRESENTATION != POST REPRESENTATION¶

So far, the representation of a programmer that we send in our PUT request exactly matches the representation the server sends in the response after a GET request. But it does't need to be this way.

When we POST or PUT to a programmer URI, we send data that contains an `avatarNumber` field. But imagine if we designed the API so that when we *GET* a programmer, the representation in the response contains an `avatarUrl` instead of `avatarNumber`. In this world, the representation we send in a request would be different than the representation that the server sends back in a response. And this would be perfectly ok!

The point is that I don't want you to feel like the data your API receives needs to look exactly like the data you send back. Nope. Both are just *representations* of a resource.

With that said, if you make the representations inconsistent for no reason, your API users will hunt you down with pitchforks. So if you expect `avatarNumber` in the POST body, don't send back `avatar_number` in the GET request. That's just mean.

Immutable Properties¶

In our API, the programmer's nickname is its unique, primary key. So, I don't really want it to be editable. In other words, even though the response representation of a programmer resource will contain a `nickname` property, a PUT request to update it should *not* have this field.

Let's first add to our scenario to test that even if we send a `nickname` field, the resource's nickname doesn't change:

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

Scenario: PUT to update a programmer
  # ...
  And I have the payload:
   """
   {
     "nickname": "CowgirlCoder",
     "avatarNumber" : 2,
     "tagLine": "foo"
   }
   """
  # ...
  But the "nickname" property should equal "CowboyCoder"
```

Run the test first to make sure it's failing. Next, let's update the `handleRequest` function to only set the `nickname` on a *new* Programmer:

```
private function handleRequest(Request $request, Programmer $programmer)
{
    $data = json_decode($request->getContent(), true);
    $isNew = !$programmer->id;

    if ($data === null) {
        throw new \Exception(sprintf('Invalid JSON: '.$request->getContent()));
    }

    // determine which properties should be changeable on this request
    $apiProperties = array('avatarNumber', 'tagLine');
    if ($isNew) {
        $apiProperties[] = 'nickname';
    }

    // ...
}
```

Now run the test:

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

Perfect! We've decided just to ignore these "extra" properties. You could also decide to return an error response instead. It just depends on your taste. What we did here is easier to use, but our client may also not notice that we're ignoring some of the submitted data. We'll talk about error responses in a few minutes.

Chapter 18: PUT Versus POST

PUT VERSUS POST¶

PUT versus POST: one of those conversations you try *not* to have. It leads to broken friendships, rainy picnics, and sad-looking kittens. People are passionate about REST, and this is one of the really sensitive topics.

First, you can read the technical descriptions in the rfc2616 document I mentioned earlier. It’s actually pretty cool stuff. But this is more than theory: you’ll need to know when to choose PUT and when to choose POST, so listen up!

Safe Methods¶

Each HTTP method is said to be “safe” or “unsafe”. An HTTP method is “safe” if using it doesn’t modify anything on the server. Ok, yes, logs will be written and analytics collected, but “safe” methods should never modify any real data. GET and HEAD are “safe” methods.

Making a request with unsafe methods - like POST, PUT and DELETE - *does* change data. Actually, if you make a request with an unsafe method it *may not* change anything. For example, if I try to update a programmer’s `avatarNumber` to the value it already has, nothing happens.

The point is that if a client uses an unsafe method, it knows that this method may have consequences. But if it uses a safe method, that request won’t ever have consequences. You could of course write an API that violates this. But that’s dishonest - like showing a picture of ice cream and then giving people broccoli. I like brocolli, but don’t be a jerk.

Being “safe” affects caching. Safe requests can be cached by a client, but unsafe requests can’t be. But caching is a whole different topic!

Idempotent¶

Within the unsafe methods, we have to talk about the famous term: “idempotency”. A request is idempotent if the side effects of making the request 1 time is the same as making the request 2, 3, 4, or 1072 times. PUT and DELETE are idempotent, POST is not.

For example, if we make the PUT request from our test once, it updates the `avatarNumber` to 2. If we make it again, the `avatarNumber` will still be 2. If we make the PUT request 1 time or 10 times, the server always results in the same state.

Now think about the POST request in our test, and imagine for a second that the `nickname` doesn’t have to be unique, because that detail clouds things here unnecessarily.

If we make the request once, it would create a programmer. If we make it again, it would create *another* programmer. So making the request 12 times is not the same as making it just once. This is *not* idempotent.

Now you can see why it *seems* right to say that POST creates resources and PUT updates them.

POST or PUT? The 2 Rules of PUT¶

Other than PATCH, which is an edge case we’ll discuss next, if you’re building an endpoint that will modify data, it should use a POST or PUT method.

Deciding between POST and PUT is easy: use PUT if and only if the endpoint will follow these 2 rules:

1. The endpoint must be idempotent: so safe to redo the request over and over again;
2. The URI must be the address to the resource being updated.

When we use PUT, we’re saying that we want the resource that we’re sending in our request to be stored at the given URI. We’re literally “putting” the resource at this address.

This is what we’re doing when we PUT to `/api/programmers/CowboyCoder`. This results in an update because `CowboyCoder` already exists. But imagine if we changed our controller code so that if `CowboyCoder` didn’t exist, it would be created. Yes, that should *still* be a PUT: we’re putting the resource at this URI. Because of this, PUT is usually thought of as an update, but it could be used to create resources. You may never choose to use PUT this way, but technically it's ok.

Heck, we *could* even support making a PUT request to `/api/programmers`. But if we did - and we followed the rules of PUT - we'd expect the client to pass us a collection of programmers, and we'd replace the existing collection with this new one.

POST and Non-Idempotency¶

One last thing. POST is not idempotent, so making a POST request more than one time *may* have additional side effects, like creating a second, third and fourth programmer. But the key word here is *may*. Just because an endpoint uses POST doesn't mean that it *must* have side effects on every request. It just *might* have side effects.

When choosing between PUT and POST, don't just say "this request is idempotent, it must be PUT!". Instead, look at the above 2 rules for put. If it fails one of those, use POST: even if the endpoint is idempotent.

Chapter 19: Deleting Resources

DELETING RESOURCES¶

After all the POST, PUT, idemptotency talk, we deserve a break. There's only one thing that an API client *can't* do to a programmer resource: delete it! So let's fix that!

Once again, we're going to leverage HTTP methods. We have GET to retrieve a representation, PUT to update the resource, and DELETE to, ya know, blow the resource up! HTTP gives us these HTTP verbs so that we don't need to do silly things like have a `/api/programmers/delete` URI. Remember, every URI is a resource, and that one wouldn't really make sense you would probably get teased in the cafeteria for it.

Writing the Test¶

Oh where to start? Why not write a test? Open up our feature file and add yet another scenario, this time for deleting a programmer resource. We need to use a `Given` like in the other scenarios to first make sure that we have a programmer in the database to delete:

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

Scenario: DELETE a programmer
  Given the following programmers exist:
    | nickname   | avatarNumber |
    | UnitTester | 3            |
  When I request "DELETE /api/programmers/UnitTester"
```

After deleting a resource, what should the endpoint return and what about the status code? People argue about this, but one common approach is to return a 204 status code, which means "No Content". It's the server's way of saying "I completed your request ok, but I really don't have anything else to tell you beyond that". In other words, the response will have an empty body:

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

Scenario: DELETE a programmer
  Given the following programmers exist:
    | nickname   | avatarNumber |
    | UnitTester | 3            |
  When I request "DELETE /api/programmers/UnitTester"
  Then the response status code should be 204
```

Coding the Endpoint¶

To make this work, we'll need to create a route that responds to the HTTP `DELETE` method. Make sure the URL is the same as what we use to GET one programmer, because we want to take the DELETE action on that resource:

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

protected function addRoutes(ControllerCollection $controllers)
{
    // ...

    $controllers->delete('/api/programmers/{nickname}', array($this, 'deleteAction'));
}
```

Next, create the `deleteAction` method. We can copy a little bit of code that queries for a programmer:

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

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

    // ...
}
```

If the programmer exists, let’s eliminate him! I’ve created a shortcut method called `delete` in my project. Your code will be different, but fortunately, deleting things is pretty easy:

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

public function deleteAction($nickname)
{
    // ...

    if ($programmer) {
        $this->delete($programmer);
    }

    // ...
}
```

And finally, we just need to send a Response back to the user. The important part is the 204 status code and the blank content, which is what 204 means:

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

public function deleteAction($nickname)
{
    // ...

    if ($programmer) {
        $this->delete($programmer);
    }

    return new Response(null, 204);
}
```

Dang, that was really easy! Execute Behat to make sure we didn’t mess anything up. Awesome! Like with everything else, be consistent with how resources are deleted. Whether you return a 204 status code, or some sort of JSON message, return the same thing for all resources when they’re deleted.

Chapter 20: PATCH: The Other Edit

PATCH: THE OTHER EDIT¶

The rules of HTTP say that in normal situations, if you want to edit a resource, you should make a PUT request to it and include the new representation you want.

Note

What are the non-normal situations? That would include custom endpoints, like an endpoint to "power up" a programmer. This will edit the programmer resource, but will have a different URI and could be POST or PUT, depending on all that idempotency stuff.

PUT is Replace¶

To get technical, PUT says "take this representation and entirely put it at this URI". It means that a REST API should require the client to send *all* of the data for a resource when updating it. If the client *doesn't* send a field, a REST API is supposed to set that field to null.

Right now, our API follows this rule. If a client sends a PUT request to `/api/programmers/ObjectOrienter` to update the `avatarNumber` field but forgets to include the `tagLine`, the `tagLine` will be set to null. Woops! There goes some perfectly good data. So PUT is really more of a *replace* than an *update*.

This is how PUT is *supposed* to work. But not all API's follow this rule, because it's a bit harsh and might cause data to be set to blank without a client intending to do that.

We're going to follow the rules and keep our PUT implementation as a replace. But how could we allow the client to update something without sending every field?

Introducing PATCH: The (Friendly) Update Method¶

With PATCH of course! The main HTTP methods like GET, POST, DELETE and PUT were introduced in the famous RFC 2616 document. Maybe you've heard of it? But because PUT has this limitation, PATCH was born in RFC 5789.

Typically, PATCH is used exactly like PUT, except if we don't send a `tagLine` field then it keeps its current value instead of obliterating it to null. PATCH, it's the friendly update.

Writing the Test¶

Let's write a scenario for PATCH, one that looks like our PUT scenario, but *only* sends the `tagLine` data in the representation. Of course, we'll want to check that the `tagLine` was updated, but also that the `avatarNumber` is unchanged:

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

Scenario: PATCH to update a programmer
  Given the following programmers exist:
    | nickname     | avatarNumber | tagLine |
    | CowboyCoder | 5             | foo      |
  And I have the payload:
    """
    {
      "tagLine": "giddyup"
    }
    """
  When I request "PATCH /api/programmers/CowboyCoder"
  Then the response status code should be 200
  And the "avatarNumber" property should equal "5"
  And the "tagLine" property should equal "giddyup"
```

Coding the Endpoint¶

Let's head into our controller and add another routing line that matches the same `/api/programmers/{nickname}` URI, but only on PATCH requests. This looks a little bit different only because Silex doesn't natively support configuring PATCH requests. But the result is the same as the other routing lines:

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

protected function addRoutes(ControllerCollection $controllers)
{
    // ...

    // point PUT and PATCH at the same controller
    $controllers->put('/api/programmers/{nickname}', array($this, 'updateAction'));

    // PATCH isn't natively supported, hence the different syntax
    $controllers->match('/api/programmers/{nickname}', array($this, 'updateAction'))
        ->method('PATCH');

    // ...
}
```

I made this route to use the same `updateAction` function as the PUT route. That's not on accident: these two HTTP methods are so similar that I bet we can save some code by re-using the controller function.

The only difference with PATCH is that if we don't see a field in the JSON that the client is sending, we should just skip updating it. Add an `if` statement that checks to see if this is a `PATCH` method and also if the field is missing from the JSON. If both are `true`, let's do nothing:

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

private function handleRequest(Request $request, Programmer $programmer)
{
    // ...
    $apiProperties = array('avatarNumber', 'tagLine');
    // ...

    foreach ($apiProperties as $property) {
        // if a property is missing on PATCH, that's ok - just skip it
        if (!isset($data[$property]) && $request->isMethod('PATCH')) {
            continue;
        }

        $val = isset($data[$property]) ? $data[$property] : null;
        $programmer->$property = $val;
    }

    // ...
}
```

And just like that, we *should* have a working PATCH endpoint. And if we somehow broke our PUT endpoint, our tests will tell us!

But we're in luck! When we run Behat, everything still comes back green. We now have 2 methods a client can use to update a resource: PUT and PATCH.

Should I Support PUT and PATCH?¶

All of this wouldn't be RESTful if it weren't a bit controversial. Because PUT's correct behavior is harsh, many APIs support PUT, but make it act like PATCH. Do what's best for your API clients, be consistent, and then make sure it's perfectly clear how things work. Remember, the more you bend the rules, the weirder your API will look when people are learning it.

The Truth Behind PATCH¶

And about PATCH, I've been lying to you. We're *technically* using PATCH incorrectly. womp womp... Let's go back to RFC 5789 where it describes PATCH with a little more detail:

> In a PUT request, the enclosed entity is considered to be a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced. With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.

Let me summarize this. With PUT, we send a representation of the resource. But with PATCH, we send a set of *instructions* on what to edit, not a representation. So instead of a JSON programmer, we might instead create some JSON structure with details on what to update:

```
[
  { "op": "replace", "path": "avatarNumber", "value": "5" },
  { "op": "remove", "path": "tagLine" }
]
```

In fact, even this little structure here comes from another proposed standard, RFC 6902. If you want to know more about this, read the blog post Please. Don't Patch Like An Idiot from this tutorial's co-author Mr. William Durand.

So what should you do in your API? It's tough, because we live in a world where the most popular API's still bend the rules. Try to follow the rules for PUT and PATCH the best you can, while still making your API very easy for your clients. And above everything, be consistent and outline your implementation in your docs.

Chapter 21: Handling Errors

HANDLING ERRORS¶

Things are looking nice, but actually, our API is kinda unusable.

We're missing a really important piece: errors! Unless we properly handle errors, our API is going to be a pain to use because the client won't know what it's doing wrong. If we deployed right now and a client tried to create a programmer with a nickname that already exists, it would get a 500 error with no details. Bummer!

Writing the Test¶

Let's start with a simple case: when a client POST's to `/api/programmers`, we should add validation to make the `nickname` field required.

Like always, let's start with the test.

We can copy the working scenario for creating a programmer, but remove the `nickname` field from the request payload. Obviously, the status code won't be 201. Because this is a *client* error, we'll need a 400-level status code. But which one? Ah, this is another spot of wonderful debate! The most common is probably 400, which means simply "Bad Request". If we look back at the RFC 2616 document, the description of the 400 status code seems to fit our situation:

> The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.

The other common choice is 422: Unprocessable Entity. 422 comes from a different RFC and is used when the format of the data is ok, but semantically, it has errors. We're adding validation for the business rule that a nickname is required, which is totally a semantic detail. So even though 422 seems to be less common than 400 for validation errors, it may be a more proper choice:

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

Scenario: Validation errors
  Given I have the payload:
    """
    {
      "avatarNumber" : "2",
      "tagLine": "I'm from a test!"
    }
    """
  When I request "POST /api/programmers"
  Then the response status code should be 400
```

Now, what should the response content look like? Obviously, it'll be JSON for our API, but let me suggest a structure that looks like this:

```
{
    "type": "validation_error",
    "title": "There was a validation error",
    "errors": {
        "nickname": "Please enter a nickname"
    }
}
```

Just trust me on the structure for now. In the test, let's look for these 3 fields and make sure we have a `nickname` error but no `avatarNumber` error.

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

Scenario: Validation errors
  Given I have the payload:
    """
    {
      "avatarNumber" : "2",
      "tagLine": "I'm from a test!"
    }
    """
  When I request "POST /api/programmers"
  Then the response status code should be 400
  And the following properties should exist:
    """
    type
    title
    errors
    """
  And the "errors.nickname" property should exist
  But the "errors.avatarNumber" property should not exist
```

Adding Validation¶

Test, check! Next, let’s actually add some validation. This is one of those spots that will be different based on your framework, but the important thing is how we *communicate* those errors to the client.

In Silex, to make the `nickname` field required, we need to open up the `Programmer` class itself. Here, add a `NotBlank` annotation with a nice message:

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

class Programmer
{
    // ...

    /**
     * @Assert\NotBlank(message="Please enter a clever nickname")
     */
    public $nickname;

    // ...
}
```

Cool! Next, open up the `ProgrammerController` class. In `newAction`, we should check the validation before saving the new `Programmer`. I’ve created a shortcut method called `validate` that does this:

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

public function newAction(Request $request)
{
    // ...
    $this->handleRequest($request, $programmer);

    $errors = $this->validate($programmer);

    $this->save($programmer);
    // ...
}
```

It uses the annotation we just added to the `Programmer` class and returns an array of errors: one error for each field. If you *are* using Silex or Symfony, you can re-use my shortcut code with your project. If you're not, just make sure you have some way of getting back an array of errors.

If the `$errors` array isn't empty, we've got a problem! And since we already wrote the test, we know *how* we want to tell the user. Create an array with the `type`, `title` and `errors` fields. The `$errors` variable is already an associative array of messages, where the keys are the field names. So we can just set it on the `errors` field:

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

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

    $errors = $this->validate($programmer);
    if (!empty($errors)) {
        $data = array(
            'type' => 'validation_error',
            'title' => 'There was a validation error',
            'errors' => $errors
        );

        return new JsonResponse($data, 400);
    }

    $this->save($programmer);
    // ...
}
```

Just like with any other API response, we can create a `JsonResponse` class and pass it our data. The only difference with this endpoint is that it has a status code of 400.

While we're here, let's move the saving of the programmer out of `handleRequest` and into `newAction` and `updateAction`:

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

    $this->handleRequest($request, $programmer);

    $errors = $this->validate($programmer);
    if (!empty($errors)) {
        // ...
    }

    $this->save($programmer);

    // ...
}

public function updateAction(Request $request, $nickname)
{
    // ... make the same change here, add $this->save($programmer);
}

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

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

This way, we can save the programmer *only* if there are *no* validation errors.

Let’s try it!

Awesome, all green!

Validation on Update¶

What’s that? You want to check our validation rules when updating too? Great idea!

To avoid duplication, create a new private function in the controller called `handleValidationResponse`. We’ll pass it an array of errors and it will transform into living robotic beings originating from the distant machine world of Cybertron err ... I mean the proper 400 JSON response:

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

private function handleValidationResponse(array $errors)
{
    $data = array(
        'type' => 'validation_error',
        'title' => 'There was a validation error',
        'errors' => $errors
    );

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

Now that we have this, use it in `newAction` and `updateAction`:

```php
// newAction and updateAction

$this->handleRequest($request, $programmer);

if ($errors = $this->validate($programmer)) {
    return $this->handleValidationResponse($errors);
}

$this->save($programmer);
```

To make sure we didn’t break anything, we can run our tests:

We could have also added another scenario to test the validation when updating. But I’m not made of scenarios people! How detailed you get with your tests is up to you.

Chapter 22: The application/problem+json Content-Type

THE APPLICATION/PROBLEM+JSON CONTENT-TYPE¶

Sometimes, there are clear rules in the API world. When we create a new resource, returning a 201 status code is the *right* thing to do.

But other times, there aren't clear rules. What we're working on right now is a good example: there's no standard for *how* API error responses should look.

Our response has `type`, `title`, and `errors` fields. And I didn't invent this: it's part of a young, potential standard called API Problem, or Problem Details. When we google for it, we find an RFC document of course! Actually, this is technically an "Internet Draft": a work-in-progress document that *may* eventually be a standard. If you use this, then you should understand that it may change in the future or be discarded entirely for something different.

But in the API world, sometimes we can choose to follow a draft like this, or nothing at all. In other words, we can choose to make our API consistent with at least *some* other API's, or consistent with noone else.

Oh, and when you're reading one of these documents, make sure you're on the latest version - they're updated all the time.

Dissecting API Problem¶

If we read a little bit, we can see that this standard outlines a response that typically has a `type` field, a `title` and sometimes a few others. The `type` field is the internal, unique identifier for an error and the title is a short human description for the `type`. If you look at our `type` and `title` values, they fit this description pretty well.

And actually, the `type` is supposed to be a URL that I can copy into my browser to get even more information about the error. Our value is just a plain string, but we'll fix that later.

The spec also allows you to add any extra fields you want. Since we need to tell the user what errors there are, we've added an `errors` field.

This means that our error response is *already* following this spec. Yay! And since someone has already defined the meaning of some of our fields, we can link to this document as part ouf or API's docs FTW!

Media Types and Structure Versus Semantics¶

Of course right now, there's no way an API client would know that we're leveraging this draft spec, unless they happen to recognize the structure. It would be much better if the response somehow screamed "I'm using the Problem Details spec!".

And this is totally possible by sending back a `Content-Type` header of `application/problem+json`. This says that the actual format is `json`, but that a client can find out more about the underlying meaning or semantics of the data by researching the `application/problem+json` Content-Type.

So the `json` part tells us how to *parse* the document. The `problem` part give us a hint on how to find out the human *meaning* of its data.

This is called the media type of the document, and if you google for IANA Media Types, you'll find a page of all of the official recognized types. You can see that there are a lot of media types that end in `+json`, like one for expressing calendar data. If you were sending calendar data, you might *choose* to use this format. Why? Because it would mean you're following a standard that is already documented, and whose structure people spent a lot of time thinking about.

Right now, I just want you to be aware that this exists, and that a lot of people invest a lot of time into answering questions like: "If 2 machines are sending calendar data, how should it be structured?".

Our `application/problem+json` actually isn't in this list, because it's just a draft.

Setting the Content-Type Header¶

But even still, we want people to know our error response is using this media type. First, let's update the test to look for this

`Content-Type` header:

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

Scenario: Validation errors
  # all the current scenario lines
  # ...
  And the "Content-Type" header should be "application/problem+json"
```

Next, add the header to our response. We’ve added plenty of response headers already, and this is no different:

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

private function handleValidationResponse(array $errors)
{
    // ...

    $response = new JsonResponse($data, 400);
    $response->headers->set('Content-Type', 'application/problem+json');

    return $response;
}
```

When we try the tests, they still pass!

And now the client knows a bit more about our error response, without us writing even one line of documentation.

Chapter 23: Enforcing Consistency with ApiProblem

ENFORCING CONSISTENCY WITH APIPROBLEM¶

We'll be returning a lot of `application/problem+json` responses, like for validation errors, 404 pages and really any error response.

And of course, I want us to *always* be consistent. To make this really easy, why not create a new `ApiProblem` class that holds all the fields we need?

Start by creating a new `Api` directory and class called `ApiProblem` :

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

class ApiProblem
{
}
```

Note

The Apigility project has a similar class, which I liked and stole the basic idea :).

By looking at the spec, I've decided that I want my problem responses to always have `status` , `type` and `title` fields, so I'll create these three properties and a `__construct` function that requires them. I'll also create a `getStatusCode` function, which we'll use in a moment:

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

class ApiProblem
{
    private $statusCode;

    private $type;

    private $title;

    public function __construct($statusCode, $type, $title)
    {
        $this->statusCode = $statusCode;
        $this->type = $type;
        $this->title = $title;
    }

    public function getStatusCode()
    {
        return $this->statusCode;
    }
}
```

Finally, since I'll need the ability to add additional fields, let's create an `$extraData` array property and a `set` function that can be used to populate it. We can use this to set the `errors` key when we're creating a validation error response:

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

class ApiProblem
{
    // ...

    private $extraData = array();

    // ...

    public function set($name, $value)
    {
        $this->extraData[$name] = $value;
    }
}
```

Back in the controller, instead of creating an array, we can now create a new `ApiProblem` object and set the data on it. This helps us enforce the structure and avoid typos:

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

private function handleValidationResponse(array $errors)
{
    $apiProblem = new ApiProblem(
        400,
        'validation_error',
        'There was a validation error'
    );
    $apiProblem->set('errors', $errors);

    // ...
}
```

Now, if we could turn the `ApiProblem` into an array, then we could just pass it to the new `JsonResponse` and be done. To do that, add a new `toArray` function to `ApiProblem`. We need to include the `type`, `title` and `status` properties as well as any extra things we set on `extraData`:

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

class ApiProblem
{
    // ...

    public function toArray()
    {
        return array_merge(
            $this->extraData,
            [
                'status' => $this->statusCode,
                'type' => $this->type,
                'title' => $this->title,
            ]
        );
    }
}
```

Cool! Use it and the `getStatusCode` function to create the `JsonResponse`:

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

private function handleValidationResponse(array $errors)
{
    // ...
    $apiProblem->set('errors', $errors);

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

    return $response;
}
```

Ok! This step made no difference to our API externally, but gave us a solid class to use for errors. This will make our code more consistent and easy to read, especially since we'll probably need to create problem responses in many places.

To try it out, just re-run the tests:

Now, just like each resource, our error responses have a PHP class that helps to model them. Very nice!

Constants: More Consistency¶

The `type` field is the unique identifier of an error, and we're supposed to have documentation for each type. So it's really important to keep track of these and never misspell them.

That sounds like a perfect use-case for constants! Add a constant on `ApiProblem` for the `validation_error` key:

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

class ApiProblem
{
    const TYPE_VALIDATION_ERROR = 'validation_error';

    // ...
}
```

Now, just reference the constant when instantiating `ApiProblem`:

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

private function handleValidationResponse(array $errors)
{
    $apiProblem = new ApiProblem(
        400,
        ApiProblem::TYPE_VALIDATION_ERROR,
        'There was a validation error'
    );

    // ...
}
```

Awesomely enough that's one less spot for me to screw up.

Mapping title to type¶

But we can go further. According to the spec, the `title` field is the description of a given `type`. In other words, we should have the exact same `title` everywhere that we use the `validation_error` `type`.

To force this consistency, create an array map on `ApiProblem` from `type` to its human-description:

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

class ApiProblem
{
    const TYPE_VALIDATION_ERROR = 'validation_error';

    static private $titles = array(
        self::TYPE_VALIDATION_ERROR => 'There was a validation error'
    );

    // ...
}
```

Note

You can also choose to translate the `title`. If you need this, you'll need to run the key through your translator before returning it.

And instead of passing the `$title` as the third argument to the constructor, we can just look it up by the `$type`. And like the good programmers we are, we'll throw a huge, ugly and descriptive exception if we don't find a title:

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

class ApiProblem
{
    // ...

    public function __construct($statusCode, $type)
    {
        $this->statusCode = $statusCode;
        $this->type = $type;

        if (!isset(self::$titles[$type])) {
            throw new \InvalidArgumentException('No title for type '.$type);
        }

        $this->title = self::$titles[$type];
    }
}
```

Back in the controller, we can now safely remove the last argument when constructing the `ApiProblem` object:

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

private function handleValidationResponse(array $errors)
{
    $apiProblem = new ApiProblem(
        400,
        ApiProblem::TYPE_VALIDATION_ERROR
    );

    // ...
}
```

Bam! We have an `ApiProblem` class to keep things consistent, a constant for the one problem `type` we have so far, and a `title` that's automatically chosen from the type.

Chapter 24: Error in Invalid JSON

ERROR IN INVALID JSON¶

Beyond validation errors, what else could go wrong? Well what if a tricky client tries to mess with us by sending invalid JSON? Right now, that would result in a cryptic 500 error message. We can't let them catch us off guard, so we need a 400 status code with a clear explanation to tell them that we're always watching.

Let's write a test! I'll copy the validation error scenario, but remove a quote so that the JSON is invalid:

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

Scenario: Error response on invalid JSON
  Given I have the payload:
    """
    {
      "avatarNumber" : "2
      "tagLine": "I'm from a test!"
    }
    """
  When I request "POST /api/programmers"
  Then the response status code should be 400
```

For now, let's just continue to check that the status code is 400. Take that shifty client! If we run the test immediately, it fails with a 500 error instead.

Handling Invalid JSON¶

In our controller, we're already checking to see if the JSON is invalid, but right now, we're throwing a normal PHP Exception message, which results in the 500 error:

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

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

    if ($data === null) {
        throw new \Exception(sprintf('Invalid JSON: '.$request->getContent());
    }

    // ...
}
```

To make this a 400 error, we could do 2 things. First, we could create a new `Response` object and set its status code to 400. That's what we're already doing with the validation error.

Second, in Silex, we could throw a special `HttpException`:

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

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

  if ($data === null) {
    throw new HttpException(
      400,
      sprintf('Invalid JSON: '.$request->getContent())
    );
  }

  // ...
}
```

In most frameworks, if you throw an exception, it results in a 500 status code. That's true in Silex too, unless you throw this very special type of exception where the status code is the first argument.

First, make sure it's working by running the test:

Tip

Silex/Symfony has a collection of exception classes, one for each of the most common 400 and 500-level responses. For example, `BadRequestHttpException` is a sub-class of `HttpException` that sets the status code to 400. The result is the same: throwing these "named" exception classes is just a bonus to give your code more consistency and clarity.

Awesome! So why am I throwing an exception instead of just returning a normal 400 response? The problem is that we're inside `handleRequest`, so if I return a `Response` object here, it won't actually be sent back to the user unless we also return that value from `newAction` and `updateAction`. That just gets confusing and a bit ugly.

Instead, if we throw an exception, the normal execution will stop immediately and the user will *definitely* get the 400 response. So being able to throw an exception like this makes my code easier to write and understand. Double threat!

The disadvantage is complexity. When I throw an exception, I need to have some other magic layer that is able to convert it into a proper response. In Silex, that magic layer is smart enough to see my `HttpException` and create a response with a 400 status code instead of 500.

If this doesn't make sense yet, keep following along.

ApiProblem for Invalid JSON¶

Since invalid JSON is a "problem", we should really send back an `application/problem+json` response. Let's first update the test to look for this `Content-Type` header and then look for a `type` field that's equal to `invalid_body_format`:

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

Scenario: Error response on invalid JSON
  # the rest of the scenario
  # ...
  And the "Content-Type" header should be "application/problem+json"
  And the "type" property should equal "invalid_body_format"
```

To make this work, we'll create a new `ApiProblem` object. But first, let's add the new `invalid_body_format` type as a constant to the class and give it a title:

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

class ApiProblem
{
    // ...
    const TYPE_INVALID_REQUEST_BODY_FORMAT = 'invalid_body_format';

    static private $titles = array(
        // ...
        self::TYPE_INVALID_REQUEST_BODY_FORMAT => 'Invalid JSON format sent',
    );

    // ...
}
```

Next, instantiate the new `ApiProblem` in the controller:

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

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

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

        // ...
    }

    // ...
}
```

But now what? When we had validation errors, we just created a new `JsonResponse`, passed `$problem->toArray()` as data, and returned it. But here, we want to throw an exception instead so that the normal flow stops.

We're going to fix this in two steps. First, we *will* throw an Exception, but we'll put the `ApiProblem` inside of it. Second, we'll hook into the magic layer that handles exceptions and extend it so that it transforms the exception into a `Response` with a 400 status code. Again, this is a little more complicated, so if it doesn't make sense yet, watch our implementation.

Chapter 25: ApiProblemException and Exception Handling

APIPROBLEMEXCEPTION AND EXCEPTION HANDLING¶

In order to be able to throw an exception that results in a JSON response, we need to hit the gym and first create a new class called `ApiProblemException`. Make it extend that special `HttpException` class:

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

use Symfony\Component\HttpKernel\Exception\HttpException;

class ApiProblemException extends HttpException
{
}
```

The purpose of this class is to act like a normal exception, but also to hold the `ApiProblem` inside of it. To do this, add an `$apiProblem` property and override the `__construct` method so that an `ApiProblem` object is the first argument:

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

use Symfony\Component\HttpKernel\Exception\HttpException;

class ApiProblemException extends HttpException
{
    private $apiProblem;

    public function __construct(ApiProblem $apiProblem, \Exception $previous = null, array $headers = array(), $code = 0
    {
        $this->apiProblem = $apiProblem;

        parent::__construct(
            $apiProblem->getStatusCode(),
            $apiProblem->getTitle(),
            $previous,
            $headers,
            $code
        );
    }
}
```

The exception still needs a message and I'm calling `getTitle()` on the `ApiProblem` object to get it. Open up the `ApiProblem` class and add this `getTitle()` function so we can access it:

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

class ApiProblem
{
    // ...

    public function getTitle()
    {
        return $this->title;
    }
}
```

Finally, go back to `ApiProblemException` and add a `getApiProblem` getter function. Hang tight, we'll use this in a few minutes:

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

use Symfony\Component\HttpKernel\Exception\HttpException;

class ApiProblemException extends HttpException
{
    // ...

    public function getApiProblem()
    {
        return $this->apiProblem;
    }
}
```

Back in the controller, throw a new `ApiProblemException` and pass the `ApiProblem` object into it:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...
use KnpU\CodeBattle\Api\ApiProblemException;
// ...

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

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

        throw new ApiProblemException($problem);
    }

    // ...
}
```

Exception Listener¶

If we run the tests now, they still fail. But notice that the status code *is* still 400. Our new exception class extends `HttpException`, so we really have the same behavior as before.

When an exception is thrown anywhere in our app, Silex catches it and gives us an opportunity to process it. In fact, this is true in just about every framework. So if you're not using Silex, just find out how to extend the exception handling in your framework and repeat what we're doing here.

Open up the `Application.php` class in the `src/KnpU/CodeBattle/` directory. This is the heart of my application, but you don't need to worry about it too much. At the bottom of the class, I've created a `configureListeners` function. By calling `$this->error`, we can pass it an anonymous function that will be called whenever there is an exception anywhere in our app. Add a debug statement so we can test it:

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

private function configureListeners()
{
    $this->error(function() {
        die('hallo!');
    });
}
```

To try it out, just open up the app in your browser and go to any 404 page, since a 404 is a type of exception:

> http://localhost:8000/foo/bar

Awesome! We see the `die` code.

Filling in the Exception Listener¶

When Silex calls the function, it passes it 2 arguments: the exception that was thrown and the status code we should use:

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

private function configureListeners()
{
    $this->error(function(\Exception $e, $statusCode) {
        die('hallo!');
    });
}
```

Tip

Silex passes a `$statusCode` argument, which is equal to the status code of the HttpException object that was thrown. If some other type of exception was thrown, it will equal 500.

Here's the cool part: if the exception is an `ApiProblemException`, then we can get the embedded `ApiProblem` object and use it to create the proper `JsonResponse`.

Let's first check for this - if it's not an `ApiProblemException`, we won't do any special processing. And if it is, we'll create the `JsonResponse` just like we might normally do in a controller:

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

private function configureListeners()
{
    $this->error(function(\Exception $e, $statusCode) {
        // only do something special if we have an ApiProblemException!
        if (!$e instanceof ApiProblemException) {
            return;
        }

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

        return $response;
    });
}
```

That's it! If we throw an `ApiProblemException`, this function will transform it into the `JsonResponse` we want. Don't believe me? Try running the tests now:

ApiProblemException for Validation¶

This is *really* powerful. If we need to return a "problem" anywhere in our API, we only need to create an `ApiProblem` object and throw an `ApiProblemException`.

Let's take advantage of this for our validation errors. Find `handleValidationResponse` and throw a new `ApiProblemException` instead of creating and returning a `JsonResponse` object. And to keep things clear, let's also rename this function to `throwApiProblemValidationException`:

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

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

    throw new ApiProblemException($apiProblem);
}
```

Now, update `newAction` and `updateAction` to use the new function name. We can also remove the `return` statements from each: we don't need that anymore:

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

// newAction() and updateAction()
if ($errors = $this->validate($programmer)) {
    $this->throwApiProblemValidationException($errors);
}
```

And when we run the tests, all green! Piece by piece, we're making our *code* more consistent so that we can guarantee that our *API* is consistent.

Chapter 26: Handling 404 Errors

HANDLING 404 ERRORS¶

We’re handling validation errors and invalid JSON errors. The last big thing is to properly handle 404 errors. In `showAction` and `updateAction`, we’re throwing a special type of exception class to trigger a 404 response. But in reality, the 404 response isn’t JSON: it’s a big HTML page. You can see this by browsing to a made-up programmer:

> http://localhost:8000/api/programmers/bumblebee

And actually, if we go to a completely made-up URL, we also see this same HTML page:

> http://localhost:8000/api/foo/bar

Internally, Silex throws that same exception to cause this 404 page.

Somehow, we need to be able to return JSON for *all* exceptions and while we are at it we should use the API problem detail format.

Writing the Test¶

First, what should we do?... anyone? Bueller? You know, write a test! Copy the GET scenario, but use a fake programmer name.

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

Scenario: Proper 404 exception on no programmer
  When I request "GET /api/programmers/fake"
  Then the response status code should be 404
  And the "Content-Type" header should be "application/problem+json"
  And the "type" property should equal "about:blank"
  And the "title" property should equal "Not Found"
```

For the `type` field, I’m going to use `about:blank`. Why? When we don’t have any extra information about an error beyond the status code, the spec says we should use this. I’m also going to check that `title` equals `Not Found`. Again, the spec says that if we use `about:blank` for `type`, then `title` should contain the standard status code’s description. 404 means “Not Found”.

Using the Exception Listener on all /api URLs¶

Now let’s roll up our sleeves and get to work! We’ll go back to the exception listener function. We want to handle *any* exception, as long as the URL starts with `/api`. We can pass a handle to this object into my anonymous function in order to get Silex’s `Request`. With it, the `getPathInfo` function gives us a clean version of the URL that we can check:

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

public function configureListeners()
{
    $app = $this;

    $this->error(function(\Exception $e, $statusCode) use ($app) {
        // only act on /api URLs
        if (strpos($app['request']->getPathInfo(), '/api') !== 0) {
            return;
        }

        // ...

        return $response;
    });
}
```

If you're not using Silex, just make sure you can check the current URL to see if it's for your API. Alternatively, you may have some other logic to know if the current request is for your API.

Always Create an ApiProblem¶

Next, we need an `ApiProblem` object so we can create our `application/problem+json` response. If the exception is an instance of `ApiProblemException`, then that's easy! If not, we need to do our best to create one:

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

$this->error(function(\Exception $e, $statusCode) use ($app) {
    // only act on /api URLs
    if (strpos($app['request']->getPathInfo(), '/api') !== 0) {
        return;
    }

    if ($e instanceof ApiProblemException) {
        $apiProblem = $e->getApiProblem();
    } else {
        $apiProblem = new ApiProblem($statusCode);
    }

    // ...
});
```

In this second case, the only information we have is the status code. This is where we should use `about:blank` as the type. But instead of doing that here, let's add a bit of logic into `ApiProblem`:

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

public function __construct($statusCode, $type = null)
{
    $this->statusCode = $statusCode;
    $this->type = $type;

    if (!$type) {
        // no type? The default is about:blank and the title should
        // be the standard status code message
        $this->type = 'about:blank';
        $this->title = isset(Response::$statusTexts[$statusCode])
            ? Response::$statusTexts[$statusCode]
            : 'Unknown HTTP status code :(';
    } else {
        if (!isset(self::$titles[$type])) {
            throw new \InvalidArgumentException('No title for type '.$type);
        }

        $this->title = self::$titles[$type];
    }
}
```

First, make `$type` optional. Then, if nothing is passed, set it to `about:blank`. Next, Silex's `Response` class has a nice map of status codes and their short description. We can use it to get a consistent title.

Back in `configureListeners`, the rest is exactly like before: use `ApiProblem` to create a `JsonResponse` and set the `application/problem+json` `Content-Type` header on it. Now, if an exception is thrown from *anywhere* in the system for a URL beginning with `/api`, the client will get back an API problem response. It took a little bit of work, but this is huge!

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

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

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

    return $response;
});
```

To make sure it's working, head back to the terminal and run the tests:

The green lights prove that even the 404 page is being transformed into a proper API problem response.

The type key should be a URL¶

We're now returning an API problem response whenever something goes wrong in our app. We can create these manually, like we did for validation errors. Or we can let them happen naturally, like when a 404 page occurs. We also have a very systematic way to create error responses, so that they stay consistent.

One last problem is that the `type` should be a URL, not just a string. One simple solution would be to prefix the `type` with the URL to some documentation page and use our code as the anchor. Let's do this inside our anonymous function, unless it's set to `about:blank`:

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

$data = $apiProblem->toArray();
if ($data['type'] != 'about:blank') {
    $data['type'] = 'http://localhost:8000/api/docs/errors#'.$data['type'];
}
$response = new JsonResponse(
    $data,
    $statusCode
);
```

Of course, creating that page is still up to you. But we’ll talk more about documentation in the next episode.

Run the tests to see if we broke anything:

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

Ah, we did! The scenario that is checking for invalid JSON is expecting the header to equal `invalid_body_format`. Tweak the scenario so the URL doesn’t break things:

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

Scenario: Error response on invalid JSON
  # ...
  And the "type" property should contain "/api/docs/errors#invalid_body_format"
```

Run the tests again. Ok, all greeen!

Chapter 27: Exposing more Error Details

EXPOSING MORE ERROR DETAILS¶

In our app, we trigger 404 pages by calling the `throw404` shortcut function in a controller. Behind the scenes, this throws a `NotFoundHttpException`, which extends that very special `HttpException` class. The result is a response with a 404 status code. That magic is part of Silex. But without this, we could have built in our own logic in the exception listener to map certain exception classes to specific status codes.

All exceptions have an optional message, which we can populate by passing an argument to `throw404`. We're already including some details that might help the API client.

If you go to `/api/programmers/bumblebee` in the browser, we get a 404 JSON response, but we don't see that message. Look back at our exception-handling function. The `NotFoundHttpException` is not an instance of `ApiProblemException`, so we fall into the situation where we create the `ApiProblem` by hand. We're not using the exception's message anywhere, so it makes sense we don't see it:

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

// ...
} else {
    $apiProblem = new ApiProblem($statusCode);
}
```

Before we fix this, let's update the 404 scenario. We can use the standard `details` field to store the exception message:

```
Scenario: Proper 404 exception on no programmer
  # ...
  And the "detail" property should equal "The programmer fake does not exist!"
```

To get this working, we're going to set the `details` property to be the exception object's message. But wait! We need to be very very careful. We don't *always* want to expose the message. What if some deep exception is thrown from the database? We might be exposing our database structure. That would be awkward.

Instead, let's *only* expose the message if the exception is an instance of `HttpException`:

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

if ($e instanceof ApiProblemException) {
    $apiProblem = $e->getApiProblem();
} else {
    $apiProblem = new ApiProblem($statusCode);

    if ($e instanceof HttpException) {
        $apiProblem->set('detail', $e->getMessage());
    }
}
```

This will now include only exceptions that are for things like 404 and 403 responses, which *we* are usually in charge of creating and throwing. If a deep database exception occurs, it won't be an `HttpException` message, so nothing will get exposed.

Run the test to try it out. Awesome! That's a much more helpful error response.

In your application, you'll still want to be careful with this. We want to be helpful to the client, but we absolutely don't want to expose any of our internals. Make sure whatever logic you use here is very solid. #security

Even *our* logic is a bit loose. For example, if we go to a URL that just doesn't exist, the client sees "No route found" in the details, which is a bit more than I want to show the user. To fix this, you could show messages from an even smaller set of exception classes. The FOSRestBundle for Symfony has a feature like this.

Big Errors When Developing¶

Let's throw a big exception from inside the `showAction` controller method and see how the tests look:

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

public function showAction($nickname)
{
    throw new \Exception('I made a mistake!');
    // ...
}
```

Let's run *just* one of the scenarios that uses `showAction`. To do this, we can point Behat directly at the `.feature` file and include the line number where the word `Scenario:` appears:

```
$ php vendor/bin/behat features/api/programmer.feature:64
```

When the test fails, it prints out what the client will see, which is our API Problem media type response... but with absolutely no details beyond the 500 status code.

While developing, that's not helpful. Instead, for 500 errors, I want to continue seeing the big beautiful, normal error page, because it includes the exception message and stacktrace.

Go back to the `Application.php` file where our exception handler function lives. Most applications have some variable that says whether you're in debug mode or not. If we *are*, and the status code is 500, let's *not* handle the exception here. Instead, the normal big error page will show:

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

$this->error(function(\Exception $e, $statusCode) use ($app) {
    // only act on /api URLs
    if (strpos($app['request']->getPathInfo(), '/api') !== 0) {
        return;
    }

    // allow 500 errors to be visible to us in debug mode
    if ($app['debug'] && $statusCode == 500) {
        return;
    }
    // ...
}
```

For Silex, there's a `debug` key on this `$app` variable, which I set in a `bootstrap.php` file. You should have something similar in your app's bootstrap or configuration. Use that! Not seeing your exception information is no fun.

Ok, be sure to remove our Exception message from `showAction` so our app works again.

Chapter 28: What’s Next

WHAT’S NEXT¶

Awesome work making it this far, seriously! We’ve covered all of the most fundamental parts of REST, and you already know enough to make a really nice API. But probably not enough to win CodeBattles. The key is to think about resources and representations, to test each part of your API, and to do things inside your code that will make your API very consistent. Be aware of the world around you and the rules of REST, but don’t be afraid to carefully bend them to make your API useable.

So what big things haven’t we covered yet? Authentication, Hypermedia and HATEOAS, Documentation, Content-Type negotiation, pagination, custom endpoints and a few other topics. Knowing these topics will go a long way to understanding the rest of the conversation about REST and to help you decide what types of things you want to use in your API, and which things might not be needed.

We’ll cover these in our next episode. Seeya next time!