Symfony RESTful API: Authentication with JWT (Course 4)

With <3 from SymfonyCasts

Chapter 1: Start Securing the App!

You again? Get outta here.... punk... is what *we* will be saying soon to API clients in this tutorial that don't have valid credentials! Yep, welcome back guys, this time to a tutorial that's making security exciting again! Seriously, I'm *pumped* to talk about authentication in an API... and in particular, a really powerful tool called *JSON web tokens*.

To make sure your JSON web tokens are the envy of all your friends, code along with me by downloading the code from any of the tutorial pages. Then, just unzip it and move into the start/ directory. I already have that start code in symfony-rest.

I also upgraded our project to Symfony 3! Woohoo! Almost everything we'll do will work for Symfony 2 or 3, but there are a few differences in the directory structure. We have a tutorial on upgrading to Symfony 3 if you want to see those.

Let's start the built-in web server with:

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

And if you just downloaded the code, open the README and follow a few other steps there.

The (sad) State of our App's Security

Ok, our app is Code Battles! It has a cool web interface and you can login with weaverryan and password foo: super secure! Here, we can create programmers and start battles. And our API already supports *a lot* of this stuff.

Open up ProgrammerController inside the Controller/Api directory:

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

```
... lines 1 - 18
19  class ProgrammerController extends BaseController
20  {
... lines 21 - 24
25      public function newAction(Request $request)
... lines 26 - 54
55      public function showAction($nickname)
... lines 56 - 76
77      public function listAction(Request $request)
... lines 78 - 95
96      public function updateAction($nickname, Request $request)
... lines 97 - 128
129     public function deleteAction($nickname)
... lines 130 - 189
190 }
```

Awesome! We can already create, fetch and update programmers. AND, we've got a pretty sweet test:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOST()
... lines 17 - 36
    public function testGETProgrammer()
... lines 38 - 59
    public function testGETProgrammerDeep()
... lines 61 - 73
    public function testGETProgrammersCollection()
... lines 75 - 91
    public function testGETProgrammersCollectionPagination()
... lines 93 - 142
    public function testPUTProgrammer()
... lines 144 - 164
    public function testPATCHProgrammer()
... lines 166 - 183
    public function testDELETEProgrammer()
... lines 185 - 246
}
```

um, suite... that checks these endpoints.

Ready for the problem? Our API has *no* security! The horror! Anonymous users are able to create programmers and then change the avatar on other programmers. It's chaos!

On the web interface, you need to be logged in to do any of these things. Let's make the API work the same way.

Testing for Security

As always: we need to start by writing a test. In ProgrammerControllerTest, add a new public function testRequiresAuthentication():

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 247
    public function testRequiresAuthentication()
    {
... lines 250 - 253
    }
}
```

Let's make an API request to an endpoint that *should* be secured and then assert some things. Start with $response = $this->client->post('/api/programmers'). Send this a valid JSON body:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 247
    public function testRequiresAuthentication()
    {
        $response = $this->client->post('/api/programmers', [
            'body' => '[]'
        ]);
... line 253
    }
}
```

Ok, if our API client tries to anonymously access a secured endpoint, what should be returned? Well, at the very least, assert that the response status code is 401, meaning "Unauthorized":

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

```
... lines 1 - 249
        $response = $this->client->post('/api/programmers', [
            'body' => '[]'
        ]);
        $this->assertEquals(401, $response->getStatusCode());
... lines 254 - 256
```

Ok! Let's go make sure this fails! Copy the method name and find the terminal. Run:

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

It fails with a validation error: it *is* getting beyond the security layer and executing our controller. Time to lock that down!

Securing a Controller

Open ProgrammerController. How can we require the API client to be authenticated? The *exact* same way you do in a web application. Add $this->denyAccessUnlessGranted('ROLE_USER'):

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

```
... lines 1 - 18
class ProgrammerController extends BaseController
{
... lines 21 - 24
    public function newAction(Request $request)
    {
        $this->denyAccessUnlessGranted('ROLE_USER');
... lines 28 - 50
    }
... lines 52 - 191
}
```

That's it. I'm using ROLE_USER because *all* of my users have this role - you could also use IS_AUTHENTICATED_FULLY.

Ok, back to the test! Run it!

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

Oh, *interesting* - it's a *200* status code instead of 401. Look closely: it redirected us to the login page. So, it's *kind* of working... you can't add programmers anonymously anymore. But clearly, we've got some work to do.

Chapter 2: JSON Web Tokens (are awesome)

How does authentication normally work on the web? Usually, after we send our username and password, a cookie is returned to us. Then, on every request after, we send that cookie back to the server: the cookie is delicious, and identifies who we are, it's our *key* to the app. The server *eats* that cookie, I mean reads that cookie, and looks it up in some database to figure out who we are.

How all (most) API Authentication Works

Guess what? An API isn't much different. One way or another, an API client will obtain a unique *token*, which - like the cookie - acts as their *key* to the API. On every request, the client will send this token and the server will use that token to figure out *who* the client is and *what* they're allowed to do.

How does it do that? Typically, the server will have a database of tokens. If I send the token I<3cookies, it can query to see if that's a valid token *and* to find out what information might be attached to it - like my user id, or even a list of permissions or *scopes*.

By the way, some of you might be wondering how OAuth fits into all of this. Well, OAuth is basically just a pattern for *how* your API client *gets* the token in the first place. And it's not the only method for obtaining tokens - we'll use a simpler method. If OAuth still befuddles you, watch our OAuth2 Tutorial. I'll mention OAuth a few more times, but mostly - stop thinking about it!

Anyways, that's token authentication in a nut shell: you pass around a secret token string instead of your username and password.

A Better way? JSON Web Tokens

But what if we could create a simpler system? What if the API client could simply send us their user ID - like 123 - on each request, instead of a token. Well, that would be awesome! Our application could just read that number, instead of needing to keep a big database of tokens and what they mean.

Alas, we can't do that. Because then *anyone* could send *any* user ID and easily authenticate as other users in the system. Right? Actually, no! We *can* do this.

In your browser, open jwt.io: the main website for JSON web tokens. These are the key to my dream. Basically, a JSON web token is nothing more than a big JSON string that contains whatever data you want to put into it - like a user's id or their favorite color. But then, the JSON is cryptographically signed and encoded to create a *new* string that doesn't look like JSON at all. This is what a JSON web token actually looks like.

But wait! JSON web tokens are encoded, but *anyone* can read them: they're easily decoded. This means their information is *not* private: you would never put something secret inside a JSON web token, like a credit card number - because - it turns out - anyone can read what's inside a JSON web token.

But here's the key: *nobody* can *modify* a JSON web token without us knowing. So if I give you a JSON web token that says your user ID is 123, someone else *could* steal this token and use it to authenticate as you. But, they *cannot* change the user ID to something else. If they do, we'll know the token has been tampered with.

That's it! JSON web tokens allow us to create tokens that actually *contain* information, instead of using random strings that require us to store and lookup the meaning of those strings on the server. It makes life simpler.

Oh, and by the way - once you eventually deploy your API, make sure it only works over SSL. No matter how you do authentication, tokens can be stolen. So, use HTTPS!

Now that we know why JSON web tokens - or JWT - rock my world, let's use them!

Chapter 3: LexikJWTAuthenticationBundle

Google for LexikJWTAuthenticationBundle. This bundle is going to make creating and validating JSON web tokens as much fun as eating ice cream. Click to read the documentation. And now, you guys know the drill. Copy the library name from the composer require line and run:

```
$ composer require lexik/jwt-authentication-bundle
```

Tip

The latest version of lexik/jwt-authentication-bundle requires Symfony 3.4 or higher. Run composer require 'lexik/jwt-authentication-bundle:v2.4' to install this bundle for older versions of Symfony, like the one that is used in this screencast. Or, upgrade your project's dependencies.

While we're waiting for Jordi, I mean Composer to download that for us, let's keep busy. Copy the new bundle line and put that into AppKernel:

53 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 20
            new Lexik\Bundle\JWTAuthenticationBundle\LexikJWTAuthenticationBundle(),
        );
... lines 23 - 32
    }
... lines 34 - 51
}
```

Great!

Generating the Public and Private Key

Our first goal is to write some code that can take an array of information - like a user's username - and turn that into a JSON web token. This bundle gives us a really handy service to do that.

But before we can use it, we need to generate a public and private key. The private, or secret key, will be used to *sign* the JSON web tokens. And no matter what the FBI says, this must stay private: if someone else gets it, they'll be able to create *new* JSON web tokens with whatever information they want - like with someone else's username to gain access to their account.

Copy the first line, head to the terminal and wait for Composer to finish all its thinking. Come on Jordi! Don't worry about the error: this bundle has some required configuration that we're *about* to provide.

First, make a new directory to hold the keys:

```
$ mkdir var/jwt
```

Next, copy the second line to create a private key, but change its path to the var/jwt directory:

```
$ openssl genrsa -out var/jwt/private.pem -aes256 4096
```

This asks you for a password - give it one! It adds another layer of security in case somebody gets your private key. I'll use

happyapi. Perfect!

Last step: copy the final line and remove app at the beginning and the end to point to the var/jwt directory:

```
$ openssl rsa -pubout -in var/jwt/private.pem -out var/jwt/public.pem
```

Type in the password you just set. This creates a *public* key. It'll be used to *verify* that a JWT hasn't been tampered with. It's not private, but you probably won't need to share it, unless someone else - or some other app - needs to *also* verify that a JWT we created is valid.

We now have a private.pem and a public.pem. You probably will *not* want to commit these to your repository: the private key needs to stay secret. But there's good news! You can create a key pair to use locally and then generate a totally different key pair on production when you deploy. They don't need to be the same. Just don't change the keys on production: that will invalidate any existing JSON web tokens that your clients have.

Configuring the Bundle

Ok, last step: tell the bundle about our keys. Copy the configuration from the docs and open up app/config/config.yml. Paste this at the bottom:

77 lines | app/config/config.yml

```
... lines 1 - 72
lexik_jwt_authentication:
    private_key_path: %kernel.root_dir%/../var/jwt/private.pem
    public_key_path:  %kernel.root_dir%/../var/jwt/public.pem
    pass_phrase:      %jwt_key_pass_phrase%
    token_ttl:        3600
```

Instead of using all these fancy parameters, it's fine to set the path directly: private_key_path: %kernel.root_dir% - that's the app/ directory - /../var/jwt/private.pem. Do the same for the public key, with public.pem. Set the token_ttl to whatever you want: I'll use 3600: this means every token will be valid for only 1 hour.

Finally, open parameters.yml and add the jwt_key_pass_phrase, which for me is happyapi. Don't forget to add an empty setting also in parameters.yml.dist for future developers:

22 lines | app/config/parameters.yml.dist

```
... line 1
parameters:
... lines 3 - 20
    jwt_key_pass_phrase: ~
```

Phew! That's it! We had to generate a public and private key, but now, life is going to be sweet. Run:

```
$ bin/console debug:container jwt
```

Select lexik_jwt_authentication.jwt_encoder. This is our new best friend for generating JSON web tokens.

Chapter 4: The "Fetch a Token" Endpoint Test

Almost every API authentication system - whether you're using JWT, OAuth or something different - works basically the same. *Somehow*, your API client gets an access token. And once it does that, it attaches it to all future requests to prove who it is and that it has access to perform some action.

How does the Client get a Token?

So, there are *two* parts to the process:

1. How the client *gets* a token
2. How a client *uses* that token.

And actually, the first part is a lot more interesting because there are a *bunch* of strategies for how a client should obtain a token. For example, you could create an endpoint where the client submits their username and password in exchange for a token. Or, you can do something more complex: like use the OAuth flow. This is a good idea when you have third-party clients - like an iPhone app - that need to gain access to your server on behalf of some user. Or, you could use both strategies - GitHub lets you do that.

But the end result is always the same: the client gets a token. We're going to build the first idea: a simple endpoint where the client can submit a username and password to get back a token. That's something that will work for most APIs.

The new Token Resource

Everything we've built so far has been centered around the Programmer resource. Now, we'll be sending back tokens: and you can think of a Token as our second API resource: the client will be able to create new tokens, and potentially, we could allow them to delete tokens.

As always, we'll start with the test. Create a new class called TokenControllerTest. Make it extend the handy ApiTestCase that we've been working on. Add public function testPOSTCreateToken():

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

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

use AppBundle\Test\ApiTestCase;

class TokenControllerTest extends ApiTestCase
{
    public function testPOSTCreateToken()
    {
... lines 11 - 20
    }
}
```

Ok, let's think about this. First, we're going to need a user in the database before we start. To create one, add $this->createUser() with weaverryan and the super-secure and realistic password I<3Pizza:

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
    public function testPOSTCreateToken()
    {
        $this->createUser('weaverryan', 'I<3Pizza');
... lines 12 - 20
    }
}
```

Next, make the POST request: $response = $this->client->post() to /api/tokens:

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 10
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
... line 14
        ]);
... lines 16 - 23
```

That URL could be anything, but the most important thing is that it's consistent with the /api/programmers we already have.

The last thing we need to do is send the username and password. And really, you can do this however you want. But, why not take advantage of the classic HTTP Basic Authentication. To send an HTTP Basic username and password with Guzzle, add an auth option and set it to an array containing the username and password:

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 10
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
            'auth' => ['weaverryan', 'I<3Pizza']
        ]);
... lines 16 - 23
```

And hey, reminder time! On production, you *will* make your API work over HTTPS. The last thing we want is plain-text password flying all over the interwebs.

Below, assert that we get back a 200 status code, or you could use 201 - since technically a resource is being created:

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 10
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
            'auth' => ['weaverryan', 'I<3Pizza']
        ]);
        $this->assertEquals(200, $response->getStatusCode());
... lines 17 - 23
```

Now, what should the response *look* like? Well, it should be a token resource... which is really just a string. Use the asserter to assert that the JSON at least contains a token property - we don't know exactly what its value will be:

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 10
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
            'auth' => ['weaverryan', 'I<3Pizza']
        ]);
        $this->assertEquals(200, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyExists(
            $response,
            'token'
        );
... lines 21 - 23
```

Looks cool! Copy the method name and run *only* this test:

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

This should fail... and it does! A 404 not found. Time to bring this to life!

Chapter 5: Create a Shiny JSON Web Token

Create a new TokenController in the Api directory:

20 lines | src/AppBundle/Controller/Api/TokenController.php

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

use AppBundle\Controller\BaseController;
... lines 6 - 9
class TokenController extends BaseController
{
... lines 12 - 19
}
```

Make this extend the same BaseController from our project and let's get to work!

First create a public function newTokenAction(). Add the @Route above and let it autocomplete so that the use statement is added for the annotation. Set the URL to /api/tokens. Heck, let's get crazy and also add @Method: we only want this route to match for POST requests:

20 lines | src/AppBundle/Controller/Api/TokenController.php

```
... lines 1 - 5
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
... lines 8 - 9
class TokenController extends BaseController
{
    /**
     * @Route("/api/tokens")
     * @Method("POST")
     */
    public function newTokenAction()
    {
... line 18
    }
}
```

To start, don't get too fancy: just return a new Response from HttpFoundation with TOKEN!:

20 lines | src/AppBundle/Controller/Api/TokenController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\Response;

class TokenController extends BaseController
{
... lines 12 - 15
    public function newTokenAction()
    {
        return new Response('TOKEN!');
    }
}
```

Got it! That won't make our test pass, but it is an improvement. Re-run it:

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

Still failing - but now it has the 200 status code.

Checking the Username and Password

Head back to TokenController. Here's the process:

1. Check that the username and password are correct.
2. Generate a JSON web token.
3. Send it back to the client.
4. High-five everyone at your office. I can't wait to get to that step.

Type-hint a new argument with Request to get the request object:

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

```
... lines 1 - 8
use Symfony\Component\HttpFoundation\Request;
... lines 10 - 12
class TokenController extends BaseController
{
... lines 15 - 18
    public function newTokenAction(Request $request)
    {
... lines 21 - 39
    }
}
```

Next, query for a User object with the normal $user = $this->getDoctrine()->getRepository('AppBundle:User') and findOneBy(['username' => '']). Get the HTTP Basic username string with $request->getUser():

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

```
... lines 1 - 12
class TokenController extends BaseController
{
... lines 15 - 18
    public function newTokenAction(Request $request)
    {
        $user = $this->getDoctrine()
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $request->getUser()]);
... lines 24 - 39
    }
}
```

And what if we can't find a user? Throw a $this->createNotFoundException():

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

```
... lines 1 - 20
        $user = $this->getDoctrine()
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $request->getUser()]);

        if (!$user) {
            throw $this->createNotFoundException();
        }
... lines 28 - 41
```

If you wanted to *hide* the fact that the username was wrong, you can throw a BadCredentialsException instead - you'll see me do that in a second.

Checking the password is *easy*: $isValid = $this->get('security.password_encoder') ->isPasswordValid(). Pass it the $user object and the raw HTTP Basic password string: $request->getPassword():

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

```
... lines 1 - 10
use Symfony\Component\Security\Core\Exception\BadCredentialsException;

class TokenController extends BaseController
{
... lines 15 - 18
    public function newTokenAction(Request $request)
    {
        $user = $this->getDoctrine()
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $request->getUser()]);

        if (!$user) {
            throw $this->createNotFoundException();
        }

        $isValid = $this->get('security.password_encoder')
            ->isPasswordValid($user, $request->getPassword());

        if (!$isValid) {
            throw new BadCredentialsException();
        }
... lines 35 - 39
    }
}
```

If this is *not* valid, throw a new BadCredentialsException. We're going to talk a lot more later about properly handling errors so that we can control the exact JSON returned. But for now, this will at least kick the user out.

Ok, ready to finally generate that JSON web token? Create a $token variable and set it to $this->get('lexik_jwt_authentication.encoder')->encode() and pass that *any* array of information you want to store in the token. Let's store ['username' => $user->getUsername()] so we know *who* this token belongs to:

44 lines | src/AppBundle/Controller/Api/TokenController.php

```
... lines 1 - 18
19      public function newTokenAction(Request $request)
20      {
21          $user = $this->getDoctrine()
22              ->getRepository('AppBundle:User')
23              ->findOneBy(['username' => $request->getUser()]);
24
... lines 25 - 35
36          $token = $this->get('lexik_jwt_authentication.encoder')
37              ->encode([
38                  'username' => $user->getUsername(),
39                  'exp' => time() + 3600 // 1 hour expiration
40              ]);
... lines 41 - 42
43      }
```

Tip

Don't forget to pass an exp key to the token, otherwise the token will *never* expire! We forgot to do this in the video!

But you can store anything here, like roles, user information, some poetry - whatever!

And that's it! This is a string, so return a new JsonResponse with a token field set to $token:

44 lines | src/AppBundle/Controller/Api/TokenController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\JsonResponse;
... lines 9 - 12
class TokenController extends BaseController
{
... lines 15 - 18
    public function newTokenAction(Request $request)
    {
        $user = $this->getDoctrine()
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $request->getUser()]);

        if (!$user) {
            throw $this->createNotFoundException();
        }

        $isValid = $this->get('security.password_encoder')
            ->isPasswordValid($user, $request->getPassword());

        if (!$isValid) {
            throw new BadCredentialsException();
        }

        $token = $this->get('lexik_jwt_authentication.encoder')
            ->encode([
                'username' => $user->getUsername(),
                'exp' => time() + 3600 // 1 hour expiration
            ]);

        return new JsonResponse(['token' => $token]);
    }
```

That's it, that's everything. Run the test!

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

It passes! Now, make sure a *bad* password fails. Duplicate this method:

23 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
    public function testPOSTCreateToken()
    {
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
            'auth' => ['weaverryan', 'I<3Pizza']
        ]);
        $this->assertEquals(200, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyExists(
            $response,
            'token'
        );
    }
}
```

and rename it to testPOSTTokenInvalidCredentials(). But now, we'll lie and pretend my password is IH8Pizza... even though we know that I<3Pizza:

33 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
... lines 9 - 22
    public function testPOSTTokenInvalidCredentials()
    {
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
            'auth' => ['weaverryan', 'IH8Pizza']
        ]);
        $this->assertEquals(401, $response->getStatusCode());
    }
}
```

Check for a 401 status code. Copy the method name and go run that test:

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

It should pass... but it doesn't! Interesting. Look at this: it definitely doesn't return the token... it redirected us to /login. We *are* getting kicked out of the controller, but this is *not* how we want our API error responses to work. We'll fix this a bit later.

Chapter 6: Authenticate a Request with JWT

We already added a denyAccessUnlessGranted() line to ProgrammerController::newAction(). That means this endpoint is broken: we don't have an API authentication system hooked up yet.

Open up ProgrammerControllerTest() and find testPOST(): the test for this endpoint:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOST()
    {
... lines 18 - 34
    }
... lines 36 - 254
}
```

Rename this to testPOSTProgrammerWorks() - this will make its name unique enough that we can run it alone:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTProgrammerWorks()
    {
... lines 18 - 40
    }
... lines 42 - 260
}
```

Copy that name and run it:

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

Instead of the 201, we get a 200 status code after being redirected to /login. I know we don't have our security system hooked up yet, but pretend that it *is* hooked up and working nicely. How can we update the test to *send* a token?

Sending a Token in the Test

Well, first, we'll need to create a valid token. Do that the same way we just did in the controller: $token = $this->getService() - which is just a shortcut we made to fetch a service from the container - and grab the lexik_jwt_authentication.encoder service. Finally, call encode() and pass it ['username' => 'weaverryan']:

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

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

        $token = $this->getService('lexik_jwt_authentication.encoder')
            ->encode(['username' => 'weaverryan']);
... lines 26 - 40
    }
... lines 42 - 260
}
```

And we have a token! Now, how do we send it to the server? Well, it's our API, so we can do whatever the heck we want! We can set it as a query string or attach it on a header. The most common way is to set it on a header called Authorization. Add a headers key to the Guzzle call with one header called Authorization. Set its value to the word Bearer, a space, and then the $token.:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTProgrammerWorks()
    {
... lines 18 - 26
        // 1) Create a programmer resource
        $response = $this->client->post('/api/programmers', [
            'body' => json_encode($data),
            'headers' => [
                'Authorization' => 'Bearer '.$token
            ]
        ]);
... lines 34 - 40
    }
... lines 42 - 260
}
```

Weird as it might look, this is a really standard way to send a token to an API. If we re-run the test now, it of course still fails. But we're finally ready to create an authentication system that looks for this token and authenticates our user.

Chapter 7: JWT Guard Authenticator (Part 1)

To create our token authentication system, we'll use Guard.

Guard is part of Symfony's core security system and makes setting up custom auth so easy it's actually fun.

Creating the Authenticator

In AppBundle, create a new Security directory. Inside add a new class: JwtTokenAuthenticator:

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 2
3  namespace AppBundle\Security;
... lines 4 - 11
12 use Symfony\Component\Security\Guard\AbstractGuardAuthenticator;
13
14 class JwtTokenAuthenticator extends AbstractGuardAuthenticator
15 {
... lines 16 - 61
62 }
```

Every authenticator starts the same way: extend AbstractGuardAuthenticator. Now, all we need to do is fill in the logic for some abstract methods. To get us started quickly, go to the "Code"->"Generate" menu - command+N on a Mac - and select "Implement Methods". Select the ones under Guard:

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 7
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
... lines 12 - 13
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
    public function getCredentials(Request $request)
    {
... lines 18 - 30
    }

    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        // TODO: Implement getUser() method.
    }

    public function checkCredentials($credentials, UserInterface $user)
    {
        // TODO: Implement checkCredentials() method.
    }

    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        // TODO: Implement onAuthenticationFailure() method.
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        // TODO: Implement onAuthenticationSuccess() method.
    }

    public function supportsRememberMe()
    {
        // TODO: Implement supportsRememberMe() method.
    }
... lines 57 - 61
}
```

Tip

Version 2 of LexikJWTAuthenticationBundle comes with an authenticator that's based off of the one we're about to build. Feel free to use it instead of building your own... once you learn how it works.

Now, do that *one* more time and also select the start() method. That'll put start() on the bottom, which will be more natural:

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 5
use Symfony\Component\HttpFoundation\Request;
... lines 7 - 8
use Symfony\Component\Security\Core\Exception\AuthenticationException;
... lines 10 - 13
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 16 - 57
    public function start(Request $request, AuthenticationException $authException = null)
    {
        // TODO: Implement start() method.
    }
}
```

If this is your first Guard authenticator... welcome to party! The process is easy: we'll walk through each method and just fill in the logic. But if you want to know more - check out the Symfony security course.

getCredentials()

First: getCredentials(). Our job is to read the Authorization header and return the token - if any - that's being passed. To help with this, we can use an object from the JWT bundle we installed earlier:
$extractor = new AuthorizationHeaderTokenExtractor(). Pass it Bearer - the prefix we're expecting before the actual token - and Authorization, the header to look on:

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 13
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
    public function getCredentials(Request $request)
    {
        $extractor = new AuthorizationHeaderTokenExtractor(
            'Bearer',
            'Authorization'
        );
... lines 22 - 30
    }
... lines 32 - 61
}
```

Grab the token with $token = $extractor–>extract() and pass it the $request:

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 17
        $extractor = new AuthorizationHeaderTokenExtractor(
            'Bearer',
            'Authorization'
        );

        $token = $extractor->extract($request);
... lines 24 - 63
```

If there is *no* token, return null:

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 17
        $extractor = new AuthorizationHeaderTokenExtractor(
            'Bearer',
            'Authorization'
        );

        $token = $extractor->extract($request);

        if (!$token) {
            return;
        }
... lines 28 - 63
```

This will cause authentication to stop. Not *fail*, just stop trying to authenticate the user via this method.

If there *is* a token, return it!

63 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 17
        $extractor = new AuthorizationHeaderTokenExtractor(
            'Bearer',
            'Authorization'
        );

        $token = $extractor->extract($request);

        if (!$token) {
            return;
        }

        return $token;
... lines 30 - 63
```

getUser()

Next, Symfony will call getUser() and pass this token string as the $credentials argument. Our job here is to use that token to find the user it relates to.

And this is where JSON web tokens really shine. Because if we simply decode the token, it will *contain* the username. Then, we can just look it up in the database.

To do this, we'll need two services. On top of the class, add a __construct() method so we can inject these. First, we need the lexik encoder service. Go back to your terminal and run:

```
$ ./bin/console debug:container lexik
```

Select the lexik_jwt_authentication.encoder service. Ah, this is just an alias for the first service - lexik_jwt_authentication.jwt_encoder. And this is an instance of JWTEncoder. Back in the authenticator, use this as the type-hint. Or wait, since it looks like there's an interface this probably implements, you can use JWTEncoderInterface instead. Give this one more argument: EntityManager $em:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 16
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 19 - 21
    public function __construct(JWTEncoderInterface $jwtEncoder, EntityManager $em)
    {
... lines 24 - 25
    }
... lines 27 - 82
}
```

I'll use a shortcut - option+enter on a Mac - to initialize these fields:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 16
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
    private $jwtEncoder;
    private $em;

    public function __construct(JWTEncoderInterface $jwtEncoder, EntityManager $em)
    {
        $this->jwtEncoder = $jwtEncoder;
        $this->em = $em;
    }
... lines 27 - 82
}
```

This created the two properties and set them for me. Nice!

Head back down to getUser(). First: decode the token. To do that, $data = $this–>jwtEncoder->decode() and pass it $credentials - that's our token string:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 16
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 19 - 43
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $data = $this->jwtEncoder->decode($credentials);
... lines 47 - 56
    }
... lines 58 - 82
}
```

That's it! $data is now an array of whatever information we originally put into the token. Fundamentally, this works just like a normal json_decode, except that the library is also checking to make sure that the contents of our token weren't changed. It does this by using our *private* key. This guarantees that nobody has changed the username to some *other* username because they're a jerk. Encryption is amazing.

It also checks the token's expiration: our tokens last 1 hour because that's what we setup in config.yml:

77 lines | app/config/config.yml

```
... lines 1 - 72
lexik_jwt_authentication:
... lines 74 - 76
    token_ttl:       3600
```

So, if ($data === false), then we know that there's a problem with the token. If there is, throw a new CustomUserMessageAuthenticationException() with Invalid token:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 45
        $data = $this->jwtEncoder->decode($credentials);

        if ($data === false) {
            throw new CustomUserMessageAuthenticationException('Invalid Token');
        }
... lines 51 - 84
```

Tip

In version 2 of the bundle, you should instead use a try-catch around this line:

```
use Lexik\Bundle\JWTAuthenticationBundle\Exception\JWTDecodeFailureException;
// ...

public function getUser($credentials, UserProviderInterface $userProvider)
{
    try {
        $data = $this->jwtEncoder->decode($credentials);
    } catch (JWTDecodeFailureException $e) {
        // if you want to, use can use $e->getReason() to find out which of the 3 possible things went wrong
        // and tweak the message accordingly
        //
https://github.com/lexik/LexikJWTAuthenticationBundle/blob/05e15967f4dab94c8a75b275692d928a2fbf6d18/Exception/JWTDecode


        throw new CustomUserMessageAuthenticationException('Invalid Token');
    }

    // ...
}
```

We'll talk about what that does in a second.

But if everything is good, get the username with $username = $data['username']:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 45
        $data = $this->jwtEncoder->decode($credentials);

        if ($data === false) {
            throw new CustomUserMessageAuthenticationException('Invalid Token');
        }

        $username = $data['username'];
... lines 53 - 84
```

Then, query for and return the user with return $this–>em–>getRepository('AppBundle:User')–>findOneBy() passing username set to $username:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 45
        $data = $this->jwtEncoder->decode($credentials);

        if ($data === false) {
            throw new CustomUserMessageAuthenticationException('Invalid Token');
        }

        $username = $data['username'];

        return $this->em
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $username]);
... lines 57 - 84
```

checkCredentials()

If the user is not found, this will return null and authentication will fail. But if a user *is* found, then Symfony finally calls checkCredentials(). Just return true:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 16
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 19 - 58
    public function checkCredentials($credentials, UserInterface $user)
    {
        return true;
    }
... lines 63 - 82
}
```

There's no password or anything else we need to check at this point.

And that's it for the important stuff!

Skip Everything Else (for now)

Skip onAuthenticationFailure() for now. And for onAuthenticationSuccess(), purposefully do nothing:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 16
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 19 - 63
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {

    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        // do nothing - let the controller be called
    }
... lines 73 - 82
}
```

We want the authenticated request to continue to the controller so we can do our normal work.

In supportsRememberMe() - this doesn't apply to us - so return false:

84 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 16
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 19 - 73
    public function supportsRememberMe()
    {
        return false;
    }
... lines 78 - 82
}
```

And keep start() blank for another minute. With just getCredentials() and getUser() filled in, our authenticator is ready to go. Let's hook it up!

Chapter 8: Registering the Authenticator (Part 2)

The authenticator class is done - well done *enough* to see it working. Next, we need to register it as a service. Open up app/config/services.yml to add it: call it jwt_token_authenticator. Set its class to AppBundle\Security\JwtTokenAuthenticator:

```
39 lines | app/config/services.yml
... lines 1 - 5
services:
... lines 7 - 35
    jwt_token_authenticator:
        class: AppBundle\Security\JwtTokenAuthenticator
        autowire: true
```

And instead of adding an arguments key: here's your permission to be lazy! Set autowire to true to make Symfony guess the arguments for us.

Finally, copy the service name and head into security.yml. Under the firewall, add a guard key, add authenticators below that and paste the service name:

```
32 lines | app/config/security.yml
security:
... lines 2 - 8
    firewalls:
        main:
... lines 11 - 20
            guard:
                authenticators:
                    - 'jwt_token_authenticator'
... lines 24 - 32
```

As *soon* as you do that, Symfony will call getCredentials() on the authenticator on *every* request. If we send a request that has an Authorization header, it should work its magic.

Let's try it! Run our original testPOSTProgrammerWorks() test: this *is* sending a valid JSON web token.

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

And this time... it passes!

Hold on, that's pretty amazing! The authenticator automatically decodes the token and authenticates the user. By the time ProgrammerController is executed, our user is logged in. In fact, there's one other spot we can *finally* fix.

Down on line 37, we originally had to make it look like *every* programmer was being created by weaverryan:

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

```
... lines 1 - 18
19  class ProgrammerController extends BaseController
20  {
... lines 21 - 24
25      public function newAction(Request $request)
26      {
... lines 27 - 36
37          $programmer->setUser($this->findUserByUsername('weaverryan'));
... lines 38 - 50
51      }
... lines 52 - 191
192 }
```

Without authentication, we didn't know *who* was actually making the API requests, and since every Programmer needs an owner, this hack was born.

Replace this with $this->getUser():

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

```
... lines 1 - 18
19  class ProgrammerController extends BaseController
20  {
... lines 21 - 24
25      public function newAction(Request $request)
26      {
... lines 27 - 36
37          $programmer->setUser($this->getUser());
... lines 38 - 50
51      }
... lines 52 - 191
192 }
```

That's it.

Our controller doesn't know or care *how* we were authenticated: it just cares that $this->getUser() returns the correct user object.

Run the test again.

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

It still passes! Welcome to our beautiful JWT authentication system. Now, time to lock down every endpoint: I don't want other users messing with my code battlers.

Chapter 9: Lock down: Require Authentication Everywhere

The *only* endpoint that requires authentication is newAction(). But to use our API, we want to require authentication to use *any* endpoint related to programmers.

Using @Security

Ok, just add $this->denyAccessUnlessGranted() to every method. OR, use a cool trick from SensioFrameworkExtraBundle. Give the controller class a doc-block and a new annotation: @Security. Auto-complete that to get the use statement. Then, add "is_granted('ROLE_USER')":

```
195 lines | src/AppBundle/Controller/Api/ProgrammerController.php
... lines 1 - 12
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
... lines 14 - 19
/**
 * @Security("is_granted('ROLE_USER')")
 */
class ProgrammerController extends BaseController
... lines 24 - 195
```

Now we're requiring a valid user on *every* endpoint.

Re-run all of the programmer tests by pointing to the file.

```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/Api/ProgrammerControllerTest.php
```

We should see a *lot* of failures. Fail, fail, fail, fail! Don't take it personally. We're *not* sending an Authorization header yet in most tests.

Sending the Authorization Header Everywhere

Let's fix that with as little work as possible. Copy the $token = code and delete it:

```
262 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTProgrammerWorks()
    {
... lines 18 - 23
        $token = $this->getService('lexik_jwt_authentication.encoder')
            ->encode(['username' => 'weaverryan']);
... lines 26 - 40
    }
... lines 42 - 260
}
```

Click into ApiTestCase and add a new protected function called getAuthorizedHeaders() with two arguments: a $username and an optional array of other $headers you want to send on the request:

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

```
... lines 1 - 20
21  class ApiTestCase extends KernelTestCase
22  {
... lines 23 - 281
282     protected function getAuthorizedHeaders($username, $headers = array())
283     {
... lines 284 - 289
290     }
... lines 291 - 345
346 }
```

Paste the $token = code here and add a new Authorization header that's equal to Bearer and then the token. Return the entire array of headers:

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

```
... lines 1 - 20
21  class ApiTestCase extends KernelTestCase
22  {
... lines 23 - 281
282     protected function getAuthorizedHeaders($username, $headers = array())
283     {
284         $token = $this->getService('lexik_jwt_authentication.encoder')
285             ->encode(['username' => $username]);
286
287         $headers['Authorization'] = 'Bearer '.$token;
288
289         return $headers;
290     }
... lines 291 - 345
346 }
```

Now, copy the method name. Oh, and don't forget to actually *use* the $username argument! In ProgrammerControllerTest, add a headers key set to $this->getAuthorizedHeaders('weaverryan'):

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

```
... lines 1 - 6
7   class ProgrammerControllerTest extends ApiTestCase
8   {
... lines 9 - 15
16      public function testPOSTProgrammerWorks()
17      {
... lines 18 - 23
24          // 1) Create a programmer resource
25          $response = $this->client->post('/api/programmers', [
... line 26
27              'headers' => $this->getAuthorizedHeaders('weaverryan')
28          ]);
... lines 29 - 35
36      }
... lines 37 - 255
256 }
```

And we just need to repeat this on every single method inside of this test. I'll look for $this->client to find these... and do it as fast as I can!

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTProgrammerWorks()
    {
... lines 18 - 24
        $response = $this->client->post('/api/programmers', [
... line 26
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 29 - 35
    }

    public function testGETProgrammer()
    {
... lines 40 - 44
        $response = $this->client->get('/api/programmers/UnitTester', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 48 - 60
    }

    public function testGETProgrammerDeep()
    {
... lines 65 - 69
        $response = $this->client->get('/api/programmers/UnitTester?deep=1', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 73 - 76
    }

    public function testGETProgrammersCollection()
    {
... lines 81 - 89
        $response = $this->client->get('/api/programmers', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 93 - 96
    }

    public function testGETProgrammersCollectionPagination()
    {
... lines 101 - 113
        $response = $this->client->get('/api/programmers?filter=programmer', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 117 - 129
        $response = $this->client->get($nextLink, [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 133 - 141
        $response = $this->client->get($lastLink, [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 145 - 153
```

```
154     }
155
156     public function testPUTProgrammer()
157     {
... lines 158 - 168
169         $response = $this->client->put('/api/programmers/CowboyCoder', [
... line 170
171             'headers' => $this->getAuthorizedHeaders('weaverryan')
172         ]);
... lines 173 - 176
177     }
178
179     public function testPATCHProgrammer()
180     {
... lines 181 - 189
190         $response = $this->client->patch('/api/programmers/CowboyCoder', [
... line 191
192             'headers' => $this->getAuthorizedHeaders('weaverryan')
193         ]);
... lines 194 - 196
197     }
198
199     public function testDELETEProgrammer()
200     {
... lines 201 - 205
206         $response = $this->client->delete('/api/programmers/UnitTester', [
207             'headers' => $this->getAuthorizedHeaders('weaverryan')
208         ]);
... line 209
210     }
211
212     public function testValidationErrors()
213     {
... lines 214 - 219
220         $response = $this->client->post('/api/programmers', [
... line 221
222             'headers' => $this->getAuthorizedHeaders('weaverryan')
223         ]);
... lines 224 - 234
235     }
236
237     public function testInvalidJson()
238     {
... lines 239 - 246
247         $response = $this->client->post('/api/programmers', [
... line 248
249             'headers' => $this->getAuthorizedHeaders('weaverryan')
250         ]);
... lines 251 - 253
254     }
255
256     public function test404Exception()
257     {
258         $response = $this->client->get('/api/programmers/fake', [
259             'headers' => $this->getAuthorizedHeaders('weaverryan')
260         ]);
```

```
    ... lines 261 - 266
267     }
    ... lines 268 - 276
277 }
```

By hooking into Guzzle, we *could* add the Authorization header to every request automatically... but there might be *some* requests where we do *not* want this header.

In fact, at the bottom, we actually test what happens when we don't send the Authorization header. Skip adding the header here:

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

```
    ... lines 1 - 6
7   class ProgrammerControllerTest extends ApiTestCase
8   {
    ... lines 9 - 268
269     public function testRequiresAuthentication()
270     {
271         $response = $this->client->post('/api/programmers', [
272             'body' => '[]'
273             // do not send auth!
274         ]);
275         $this->assertEquals(401, $response->getStatusCode());
276     }
277 }
```

With any luck, we should get a bunch of *beautiful* passes.

```
$ ./vendor/bin/phpunit tests/AppBundle/Controller/Api/ProgrammerControllerTest.php
```

And we do! Ooh, until we hit the last test! When we *don't* send an Authorization header to an endpoint that requires authentication... it's *still* returning a 200 status code instead of 401. When we kick out non-authenticated API requests, they are *still* being redirected to the login page... which is clearly *not* a cool way for an API to behave.

Time to fix that.

Chapter 10: The "Entry Point" & Multiple Firewalls

The authentication system works great! Except for how it behaves when things go wrong. When an API client tries to access a protected endpoint but forgets to send an Authorization header, they're redirected to the login page. But, why?

Here's what's going on. Whenever an anonymous user comes into a Symfony app and tries to access a protected page, Symfony triggers something called an "entry point". Basically, Symfony wants to be super hip and helpful by *instructing* the user that they need to login. In a traditional HTML form app, that means redirecting the user to the login page.

But in an api, we instruct the API client that credentials are needed by returning a 401 response. So, how can we control this entry point? In Guard authentication, you control it with the start() method.

The start() Method

Return a new JsonResponse and we'll just say error => 'auth required' as a start. Then, set the status code to 401:

90 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 17
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 20 - 79
    public function start(Request $request, AuthenticationException $authException = null)
    {
        // called when authentication info is missing from a
        // request that requires it

        return new JsonResponse([
            'error' => 'auth required'
        ], 401);
    }
}
```

To see if it's working, copy the testRequiresAuthentication method name and run that test:

```
./vendor/bin/phpunit --filter testRequiresAuthentication
```

Huh, it didn't change *anything*: we're still redirected to the login page. I thought Symfony was supposed to call our start() method in this situation? So what gives?

One Entry Point per Firewall

Open up security.yml:

32 lines | app/config/security.yml

```
security:
... lines 2 - 8
    firewalls:
        main:
            pattern: ^/
            anonymous: true
            form_login:
                # The route name that the login form submits to
                check_path: security_login_check
                login_path: security_login_form

            logout:
                # The route name the user can go to in order to logout
                path: security_logout
            guard:
                authenticators:
                    - 'jwt_token_authenticator'
... lines 24 - 32
```

Here's the problem: we have a single firewall. When an anonymous request accesses the site and hits a page that requires a valid user, Symfony has to figure out what *one* thing to do. If this were a traditional app, we should redirect the user to /login. If this were an API, we should return a 401 response. But our app is *both*: we have an HTML frontend and API endpoints. Symfony doesn't really know what *one* thing to do.

32 lines | app/config/security.yml

```
security:
... lines 2 - 8
    firewalls:
        main:
... lines 11 - 12
            form_login:
                # The route name that the login form submits to
                check_path: security_login_check
                login_path: security_login_form
... lines 17 - 32
```

The form_login authentication mechanism has a built-in entry point and *it* is taking priority. Our cute start() entry point function is being totally ignored.

But no worries, you can control this! You could add an entry_point key under your firewall and point to the authenticator service to say "No no no: I want to use my authenticator as the *one* entry point". But then, our HTML app would break: we *still* want users on the frontend to be redirected.

Normally, I'm a big advocate of having a single firewall. But this is a *perfect* use-case for splitting into two firewalls: we really do have two very different authentication systems at work.

Adding the Second Firewall

Above, the main firewall, add a new key called api: the name is not important. And set pattern: ^/api/:

36 lines | app/config/security.yml

```
security:
... lines 2 - 8
    firewalls:
        api:
            pattern: ^/api/
... lines 12 - 36
```

That's a regular expression, so it'll match anything starting with /api/. Oh, and when Symfony boots, it only matches and uses *one* firewall. Going to /api/something will use the api firewall. Everything else will match the main firewall. And this is *exactly* what we want.

Add the anonymous key: we may still want some endpoints to not require authentication:

36 lines | app/config/security.yml

```
security:
... lines 2 - 8
    firewalls:
        api:
            pattern: ^/api/
            anonymous: true
            stateless: true
... lines 14 - 36
```

I'll also add stateless: true. This is kind of cool: it tells Symfony to *not* store the user in the session. That's perfect: we expect the client to send a valid Authorization header on *every* request.

Move the guard authenticator up into the api firewall:

36 lines | app/config/security.yml

```
security:
... lines 2 - 8
    firewalls:
        api:
            pattern: ^/api/
            anonymous: true
            stateless: true
            guard:
                authenticators:
                    - 'jwt_token_authenticator'
        main:
            pattern: ^/
            anonymous: true
            form_login:
                # The route name that the login form submits to
                check_path: security_login_check
                login_path: security_login_form

            logout:
                # The route name the user can go to in order to logout
                path: security_logout
... lines 28 - 36
```

And that should do it! Now, it will use the start() method from *our* authenticator.

Give it a try!

```
./vendor/bin/phpunit –filter testRequiresAuthentication
```

It passes! Don't rush into having multiple firewalls, but if you have two very different ways of authentication, it *could* be useful.

Chapter 11: JSON Errors in your API

Whenever something goes wrong in our API, we have a *great* setup: we always get back a descriptive JSON structure with keys that describe what went wrong:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 255
    public function test404Exception()
    {
... lines 258 - 261
        $this->assertEquals(404, $response->getStatusCode());
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type')[0]);
        $this->asserter()->assertResponsePropertyEquals($response, 'type', 'about:blank');
        $this->asserter()->assertResponsePropertyEquals($response, 'title', 'Not Found');
        $this->asserter()->assertResponsePropertyEquals($response, 'detail', 'No programmer found with nickname "fake"');
    }
... lines 268 - 276
}
```

I want to do the exact same thing when something goes wrong with authentication.

Open up the TokenControllerTest:

33 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
... lines 9 - 22
    public function testPOSTTokenInvalidCredentials()
    {
        $this->createUser('weaverryan', 'I<3Pizza');

        $response = $this->client->post('/api/tokens', [
            'auth' => ['weaverryan', 'IH8Pizza']
        ]);
        $this->assertEquals(401, $response->getStatusCode());
    }
}
```

Here, we purposefully send an *invalid* username and password combination. This actually hits TokenController, we throw this new BadCredentialsException and that kicks us out:

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

```
... lines 1 - 12
class TokenController extends BaseController
{
... lines 15 - 18
    public function newTokenAction(Request $request)
    {
... lines 21 - 31
        if (!$isValid) {
            throw new BadCredentialsException();
        }
... lines 35 - 39
    }
}
```

It turns out that doing this this *also* triggers the entry point. And if you think about it, that makes sense: any time an anonymous user is able to get into your application:

90 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 17
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 20 - 79
    public function start(Request $request, AuthenticationException $authException = null)
    {
        // called when authentication info is missing from a
        // request that requires it

        return new JsonResponse([
            'error' => 'auth required'
        ], 401);
    }
}
```

And then you throw an exception to deny access, that will trigger the entry point. And our entry point is *not* yet returning the nice API problem structure.

Testing for the API Problem Response

Copy the last four lines from one of the tests in ProgrammerControllerTest:

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 255
    public function test404Exception()
    {
... lines 258 - 262
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type')[0]);
        $this->asserter()->assertResponsePropertyEquals($response, 'type', 'about:blank');
        $this->asserter()->assertResponsePropertyEquals($response, 'title', 'Not Found');
        $this->asserter()->assertResponsePropertyEquals($response, 'detail', 'No programmer found with nickname "fake"');
    }
... lines 268 - 276
}
```

And add that to testPostTokenInvalidCredentials():

37 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
... lines 9 - 22
    public function testPOSTTokenInvalidCredentials()
    {
... lines 25 - 29
        $this->assertEquals(401, $response->getStatusCode());
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type')[0]);
        $this->asserter()->assertResponsePropertyEquals($response, 'type', 'about:blank');
        $this->asserter()->assertResponsePropertyEquals($response, 'title', 'Unauthorized');
        $this->asserter()->assertResponsePropertyEquals($response, 'detail', 'Invalid credentials.');
    }
}
```

The header should be application/problem+json. The type should be about:blank: that's what you should use when the status code - 401 here - already fully describes what went wrong. For the title use Unauthorized - that's the standard text that always goes with a 401 status code. The ApiProblem class will actually set that for us: when we pass a null type, it sets type to about:blank and looks up the correct title.

Finally, for detail - which is an optional field for an API problem response - use Invalid Credentials. with a period. I'll show you *why* we're expecting that in a second.

ApiProblem in start()

Head to the JwtTokenAuthenticator. In start(), create a new $apiProblem = new ApiProblem(). Pass it a 401 status code with no type:

94 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 18
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 21 - 80
    public function start(Request $request, AuthenticationException $authException = null)
    {
        // called when authentication info is missing from a
        // request that requires it

        $apiProblem = new ApiProblem(401);
... lines 87 - 91
    }
}
```

The detail key should tell the API client *any* other information about what went wrong. And check this out: when the start() method is called, it has an optional $authException argument. Most of the time, when Symfony calls start() its because an AuthenticationException has been thrown. And *this* class gives us some information about *what* caused this situation.

And in fact, in TokenController, we're throwing a BadCredentialsException, which is a sub-class of AuthenticationException. Hold command to look inside the class:

30 lines | vendor/symfony/symfony/src/Symfony/Component/Security/Core/Exception/BadCredentialsException.php

```
... lines 1 - 19
class BadCredentialsException extends AuthenticationException
{
... lines 22 - 24
    public function getMessageKey()
    {
        return 'Invalid credentials.';
    }
}
```

It has a getMessageKey() method set to Invalid Credentials.: make sure you test matches this string exactly:

37 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
... lines 9 - 22
    public function testPOSTTokenInvalidCredentials()
    {
... lines 25 - 33
        $this->asserter()->assertResponsePropertyEquals($response, 'detail', 'Invalid credentials.');
    }
}
```

The AuthenticationException - and its sub-classes - are special: each has a getMessageKey() method that you can safely return to the user to help *hint* as to what went wrong.

Add $message = $authException ? $authException->getMessageKey() : 'Missing Credentials';:

94 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 18
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 21 - 80
    public function start(Request $request, AuthenticationException $authException = null)
    {
        // called when authentication info is missing from a
        // request that requires it

        $apiProblem = new ApiProblem(401);
        // you could translate this
        $message = $authException ? $authException->getMessageKey() : 'Missing credentials';
... lines 89 - 91
    }
}
```

If no $authException is passed, this is the best message we can return to the client. Finish this with $apiProblem->set('details', $message).:

94 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 82
        // called when authentication info is missing from a
        // request that requires it

        $apiProblem = new ApiProblem(401);
        // you could translate this
        $message = $authException ? $authException->getMessageKey() : 'Missing credentials';
        $apiProblem->set('detail', $message);
... lines 90 - 94
```

Finally, return a new JsonResponse with $apiProblem->toArray() and then a 401:

94 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 18
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 21 - 80
    public function start(Request $request, AuthenticationException $authException = null)
    {
        // called when authentication info is missing from a
        // request that requires it

        $apiProblem = new ApiProblem(401);
        // you could translate this
        $message = $authException ? $authException->getMessageKey() : 'Missing credentials';
        $apiProblem->set('detail', $message);

        return new JsonResponse($apiProblem->toArray(), 401);
    }
}
```

Perfect! Well, not *actually* perfect, but it's getting close.

Copy the invalid credentials test method and run:

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

It's close! The response looks right, but the Content-Type header is application/json instead of the more descriptive application/problem+json.

Well that's no problem! We just need to set the header inside of the start() method. But wait! Don't do that! Because we've done all of this work before.

Chapter 12: ResponseFactory: Centralize Error Responses

In the EventListener directory, we created an ApiExceptionSubscriber whose job is to catch all exceptions and turn them into nice API problem responses. And it already has all of the logic we need to turn an ApiProblem object into a proper response:

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

```
... lines 1 - 12
class ApiExceptionSubscriber implements EventSubscriberInterface
{
... lines 15 - 21
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 24 - 70
    }

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

Instead of re-doing this in the authenticator, let's centralize and re-use this stuff! Copy the last ten lines or so out of ApiExceptionSubscriber:

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

```
... lines 1 - 12
class ApiExceptionSubscriber implements EventSubscriberInterface
{
... lines 15 - 21
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 24 - 57
        $data = $apiProblem->toArray();
        // making type a URL, to a temporarily fake page
        if ($data['type'] != 'about:blank') {
            $data['type'] = 'http://localhost:8000/docs/errors#'.$data['type'];
        }

        $response = new JsonResponse(
            $data,
            $apiProblem->getStatusCode()
        );
        $response->headers->set('Content-Type', 'application/problem+json');
... lines 69 - 70
    }
... lines 72 - 78
}
```

And in the Api directory, create a new class called ResponseFactory. Inside, give this a public function called createResponse(). We'll pass it the ApiProblem and *it* will turn that into a JsonResponse:

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

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

use Symfony\Component\HttpFoundation\JsonResponse;

class ResponseFactory
{
    public function createResponse(ApiProblem $apiProblem)
    {
        $data = $apiProblem->toArray();
        // making type a URL, to a temporarily fake page
        if ($data['type'] != 'about:blank') {
            $data['type'] = 'http://localhost:8000/docs/errors#'.$data['type'];
        }

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

        return $response;
    }
}
```

Perfect! Next, go into services.yml and register this: how about api.response_factory. Set the class to AppBundle\Api\ResponseFactory and leave off the arguments key:

42 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 39
    api.response_factory:
        class: AppBundle\Api\ResponseFactory
```

Using the new ResponseFactory

We will *definitely* need this inside ApiExceptionSubscriber, so add it as a second argument: @api.response_factory:

42 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 19
    api_exception_subscriber:
... line 21
        arguments: ['%kernel.debug%', '@api.response_factory']
... lines 23 - 42
```

In the class, add the second constructor argument. I'll use option+enter to quickly create that property and set it for me:

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

```
... lines 1 - 13
class ApiExceptionSubscriber implements EventSubscriberInterface
{
... line 16
    private $responseFactory;

    public function __construct($debug, ResponseFactory $responseFactory)
    {
... line 21
        $this->responseFactory = $responseFactory;
    }
... lines 24 - 71
}
```

Below, it's very simple: $response = $this->responseFactory->createResponse() and pass it $apiProblem:

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

```
... lines 1 - 13
class ApiExceptionSubscriber implements EventSubscriberInterface
{
... lines 16 - 24
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
... lines 27 - 60
        $response = $this->responseFactory->createResponse($apiProblem);
... lines 62 - 63
    }
... lines 65 - 71
}
```

LOVE it. Let's celebrate by doing the same in the authenticator. Add a third constructor argument and then create the property and set it:

97 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 23
    private $responseFactory;

    public function __construct(JWTEncoderInterface $jwtEncoder, EntityManager $em, ResponseFactory $responseFactory)
    {
... lines 28 - 29
        $this->responseFactory = $responseFactory;
    }
... lines 32 - 95
}
```

Down in start(), return $this->responseFactory->createResponse() and pass it $apiProblem:

97 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 83
    public function start(Request $request, AuthenticationException $authException = null)
    {
... lines 86 - 93
        return $this->responseFactory->createResponse($apiProblem);
    }
}
```

Finally, go back to services.yml to update the arguments. Just kidding! We're using autowiring, so it will automatically add the third argument for us:

42 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 35
    jwt_token_authenticator:
... line 37
        autowire: true
... lines 39 - 42
```

If everything went well, we should be able to re-run the test with great success:

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

detail(s) Make tests Fails

Oh, boy - it failed. Let's see - something is wrong with the detail field:

> Error reading property detail from available keys detail**s**.

That sounds like a Ryan mistake! Open up TokenControllerTest: the test is looking for detail - with *no* s:

37 lines | tests/AppBundle/Controller/Api/TokenControllerTest.php

```
... lines 1 - 6
class TokenControllerTest extends ApiTestCase
{
... lines 9 - 22
    public function testPOSTTokenInvalidCredentials()
    {
... lines 25 - 33
        $this->asserter()->assertResponsePropertyEquals($response, 'detail', 'Invalid credentials.');
    }
}
```

That's correct. Inside JwtTokenAuthenticator, change that key to detail:

97 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 83
    public function start(Request $request, AuthenticationException $authException = null)
    {
... lines 86 - 91
        $apiProblem->set('detail', $message);
... lines 93 - 94
    }
}
```

Ok, technically we can call this field whatever we want, but detail is kind of a standard.

Try the test again.

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

That looks perfect. In fact, run our *entire* test suite:

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

Hey! We didn't break any of our existing error handling. Awesome!

But there is *one* more case we haven't covered: what happens if somebody sends a *bad* JSON web token - maybe it's expired. Let's handle that final case next.

Chapter 13: Graceful Errors for an Invalid JWT

We already know that if the client *forgets* to send a token, Symfony calls the start() method:

97 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 83
    public function start(Request $request, AuthenticationException $authException = null)
    {
... lines 86 - 94
    }
}
```

But what happens if authentication fails?

Testing with a bad Token

Let's find out! Copy testRequiresAuthentication(), paste it, and rename it to testBadToken():

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 277
    public function testBadToken()
    {
        $response = $this->client->post('/api/programmers', [
            'body' => '[]',
            'headers' => [
                'Authorization' => 'Bearer WRONG'
            ]
        ]);
        $this->assertEquals(401, $response->getStatusCode());
        $this->assertEquals('application/problem+json', $response->getHeader('Content-Type')[0]);
    }
}
```

In this case, we *will* add a headers key and we *will* send an Authorization header... but set to Bearer WRONG.

If this happens, we definitely want a 401 status code and - like always - an application/problem+json response header. Let's *just* look for these two things for now.

How Authentication Fails

When JWT authentication fails, what handles that? Well, onAuthenticationFailure() of course:

97 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 68
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {

    }
... lines 73 - 95
}
```

The getUser() method *must* return a User object. If it doesn't, then onAuthenticationFailure() is called. In our case, there are two possible reasons: the token might be corrupted or expired *or* - somehow - the decoded username doesn't exist in our database. In both cases, we are *not* returning a User object, and this triggers onAuthenticationFailure().

To start, just return a new JsonResponse that says Hello, but with the proper 401 status code:

97 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 68
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        return new JsonResponse('Hello!', 401);
    }
... lines 73 - 95
}
```

Copy the testBadToken method name and give it a try!

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

ApiProblem on Failure

It *almost* works - that's a good start. It proves our code in onAuthenticationFailure() is handling things. Now, let's setup a proper API problem response, just like we did before: $apiProblem = new ApiProblem with a 401 status code:

101 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 68
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        $apiProblem = new ApiProblem(401);
... lines 72 - 75
    }
... lines 77 - 99
}
```

Then, use $apiProblem->set() to add a detail field. And in this case, we *always* have an AuthenticationException that can hint what went wrong. Use its getMessageKey() method:

101 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 70
        $apiProblem = new ApiProblem(401);
        // you could translate this
        $apiProblem->set('detail', $exception->getMessageKey());
... lines 74 - 101
```

Oh, and by the way - if you want, you can send this through the translator service and translate into multiple languages.

Finish this with return $this–>responseFactory->createResponse() to turn the $apiProblem into a nice JSON response:

101 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 70
        $apiProblem = new ApiProblem(401);
        // you could translate this
        $apiProblem->set('detail', $exception->getMessageKey());

        return $this->responseFactory->createResponse($apiProblem);
... lines 76 - 101
```

That's it! We did all the hard work earlier.

I want to actually *see* how this response looks. So, add a $this->debugResponse() at the end of testBadToken():

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

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 277
    public function testBadToken()
    {
... lines 280 - 287
        $this->debugResponse($response);
    }
}
```

Now, re-run the test!

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

Check that out - it's beautiful! It has all the fields it needs, including detail, which is set to Invalid token.

Controlling Error Message

That text is coming from *our* code, when we throw the CustomUserMessageAuthenticationException. The text - Invalid token - becomes the "message key" and this exception is passed to onAuthenticationFailure().

This gives you complete control over how your errors look.

Chapter 14: JWT: Other Things to Think about

Mostly, that's it! JWT authentication is pretty cool: create an endpoint to *fetch* a token and an authenticator to check if that token is valid. With the error handling we added, this is a *really* robust system.

But, there are a few other things I want you to think about: things that *you* may want to consider for your situation.

Adding Scopes/Roles to the Token

First, when we created our JWT, we put the username inside of it:

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

```
... lines 1 - 12
class TokenController extends BaseController
{
... lines 15 - 18
    public function newTokenAction(Request $request)
    {
        $user = $this->getDoctrine()
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $request->getUser()]);

        if (!$user) {
            throw $this->createNotFoundException();
        }

        $isValid = $this->get('security.password_encoder')
            ->isPasswordValid($user, $request->getPassword());

        if (!$isValid) {
            throw new BadCredentialsException();
        }

        $token = $this->get('lexik_jwt_authentication.encoder')
            ->encode(['username' => $user->getUsername()]);

        return new JsonResponse(['token' => $token]);
    }
}
```

Later, we used that to query for the User object:

101 lines | src/AppBundle/Security/JwtTokenAuthenticator.php

```
... lines 1 - 19
class JwtTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 22 - 48
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $data = $this->jwtEncoder->decode($credentials);

        if ($data === false) {
            throw new CustomUserMessageAuthenticationException('Invalid Token');
        }

        $username = $data['username'];

        return $this->em
            ->getRepository('AppBundle:User')
            ->findOneBy(['username' => $username]);
    }
... lines 63 - 99
}
```

But, you can put *any* information in your token. In fact, you could also include "scopes" - or "roles" to use a more Symfony-ish word - inside your token. Also, nobody is forcing your authenticator to load a user from the database. To get really crazy, you could decode the token and create some new, non-entity User object, and populate it entirely from the information inside of that token.

And really, not everyone issues tokens that are related to a specific user in their system. Sometimes, tokens are more like a package of permissions that describe what an API client can and can't do. This is a *powerful* idea.

OAuth versus JWT

And what about OAuth? If you've watched our OAuth tutorial, then you remember that OAuth is just a mechanism for securely delivering a token to an API client. You may or may not need OAuth for your app, but if you *do* use it, you still have the option to use JSON web tokens as your bearer, or access tokens. It's not an OAuth versus JWT thing: each accomplishes different goals.

Refresh Tokens

Finally, let's talk refresh tokens. In our app, we gave our tokens a lifetime of 1 hour:

77 lines | app/config/config.yml

```
... lines 1 - 72
lexik_jwt_authentication:
... lines 74 - 76
    token_ttl:       3600
```

You see, JWT's aren't supposed to last forever. If you need them to, you might choose to issue a refresh token along with your normal access token. Then later, an API client could send the refresh token to the server and exchange it for a new JWT access token. Implementing this is pretty easy: it involves creating an extra token and an endpoint for exchanging it later. Auth0 - a leader in JWT - has a nice blog post about it.

Ok! If you have any questions, let me know. I know this stuff can be crazy confusing! Do your best to not overcomplicate things.

And as always, I'll see you guys next time.