KnpUGuard: Symfony Authentication with a Smile

With <3 from KnpUniversity

Chapter 1: Installation

Installation is easy! So let's get it behind us!

1) Install the Library via Composer

Download Composer and then run this command from inside your project.

```
$ composer require knpuniversity/guard-bundle:~0.1@dev
```

Use `php composer.phar require knpuniversity/guard-bundle:~0.1@dev` if you don't have Composer installed globally.

2) Enable the Bundle

Find your `app/AppKernel.php` file and enable the bundle:

```
39 lines | app/AppKernel.php
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 19
            new KnpU\GuardBundle\KnpUGuardBundle()
        );
... lines 22 - 31
    }
... lines 33 - 37
}
```

3) Build your first authenticator!

You're ready to build your authentication system!

A) Learn the fundamentals by building a login form

B) Create an API token authentication system

Chapter 2: How to Create a Login Form

So you want a login form? That's simple. And along the way, you'll learn all the steps that happen during authentication, and how you can customize what happens at each one.

You still have to do some work, but you're going to be really happy with the result.

Tip

Click **Download** to get the starting or finished code of this tutorial.

Create the Login Form

Don't think about security yet! Instead, start by creating a Symfony controller with two action methods: one for rendering the login form and another that'll handle the login submit:

33 lines | src/AppBundle/Controller/SecurityController.php

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="security_login")
     */
    public function loginAction()
    {
        $helper = $this->get('security.authentication_utils');

        return $this->render('security/login.html.twig', array(
            // last username entered by the user (if any)
            'last_username' => $helper->getLastUsername(),
            // last authentication error (if any)
            'error' => $helper->getLastAuthenticationError(),
        ));
    }

    /**
     * @Route("/login_check", name="security_login_check")
     */
    public function loginCheckAction()
    {
        // will never be executed
    }
}
```

So far, this is just a lovely, but boring set of actions. The only interesting parts are the `last_username` and `error` variables. Where are those coming from? You'll see. Also, `loginCheckAction()` doesn't do anything - and it never will. Another layer will handle the login submit.

Next, create the login template:

25 lines | app/Resources/views/security/login.html.twig

```
{% extends 'base.html.twig' %}

{% block body %}
  <form action="{{ path('security_login_check') }}" method="post">
    {% if error %}
      <div class="alert alert-danger">
        {{ error.messageKey|trans(error.messageData) }}
      </div>
    {% endif %}

    <div>
      <label for="username">Username</label>
      <input type="text" id="username" name="_username" value="{{ last_username }}" />
    </div>

    <div>
      <label for="password">Password:</label>
      <input type="password" id="password" name="_password" />
    </div>

    <br/>
    <button type="submit">Login</button>
  </form>
{% endblock %}
```

This form submits to the `/login_check` URL and the field names are `_username` and `_password`. Remember these - they'll be important in a minute (see getCredentials()).

Installing Guard

Read the short Installation chapter to make sure you've got the bundle installed and enabled.

Creating an Authenticator

With Guard, the whole authentication process - fetching the username/password POST values, validating the password, redirecting after success, etc - is handled in a single class called an "Authenticator". Your authenticator can be as crazy as you want, as long as it implements KnpU\Guard\GuardAuthenticatorInterface.

For login forms, life is easier, thanks to a convenience class called `AbstractFormLoginAuthenticator`. Create a new `FormLoginAuthenticator` class, make it extend this class, and add all the missing methods (from the interface and abstract class):

```
37 lines | src/AppBundle/Security/FormLoginAuthenticator.php
```

```php
// ... lines 1 - 2
namespace AppBundle\Security;

use KnpU\Guard\Authenticator\AbstractFormLoginAuthenticator;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;

class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
    public function getCredentials(Request $request)
    {
        // TODO: Implement getCredentials() method.
    }

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

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

    protected function getLoginUrl()
    {
        // TODO: Implement getLoginUrl() method.
    }

    protected function getDefaultSuccessRedirectUrl()
    {
        // TODO: Implement getDefaultSuccessRedirectUrl() method.
    }
}
```

Your mission: fill in each method. We'll get to that in a second.

To fill in those methods, we're going to need some services. To keep this tutorial simple, let's pass the entire container into our authenticator:

45 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 5
use Symfony\Component\DependencyInjection\ContainerInterface;
// ... lines 7 - 10
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
    private $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }
// ... lines 19 - 43
}
```

Tip

For seasoned-Symfony devs, you can of course inject *only* the services you need.

Registering your Authenticator

Before filling in the methods, let's tell Symfony about our fancy new authenticator. First, register it as a service:

10 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
    app.form_login_authenticator:
        class: AppBundle\Security\FormLoginAuthenticator
        arguments: ["@service_container"]
```

Next, update your `security.yml` file to use the new service:

24 lines | app/config/security.yml

```yaml
security:
... lines 2 - 13
    firewalls:
... lines 15 - 18
        main:
            anonymous: ~
            knpu_guard:
                authenticators:
                    - app.form_login_authenticator
```

Your firewall (called `main` here) can look however you want, as long as it has a `knpu_guard` section under it with an `authenticators` key that includes the service name that you setup a second ago (`app.form_login_authenticator` in my example).

I've also setup my "user provider" to load my users from the database:

24 lines | app/config/security.yml

```yaml
security:

    encoders:
        # Our user class and the algorithm we'll use to encode passwords
        # http://symfony.com/doc/current/book/security.html#encoding-the-user-s-password
        AppBundle\Entity\User: bcrypt

    providers:
        # Simple example of loading users via Doctrine
        # To load users from somewhere else: http://symfony.com/doc/current/cookbook/security/custom_provider.html
        database_users:
            entity: { class: AppBundle:User, property: username }
... lines 13 - 24
```

In a minute, you'll see where that's used.

Filling in the Authenticator Methods

Your authenticator is now being used by Symfony. So let's fill in each method:

getCredentials()

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 6
use Symfony\Component\HttpFoundation\Request;
// ... lines 8 - 9
use Symfony\Component\Security\Core\Security;
// ... lines 11 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
// ... lines 16 - 22
    public function getCredentials(Request $request)
    {
        if ($request->getPathInfo() != '/login_check') {
            return;
        }

        $username = $request->request->get('_username');
        $request->getSession()->set(Security::LAST_USERNAME, $username);
        $password = $request->request->get('_password');

        return array(
            'username' => $username,
            'password' => $password
        );
    }
// ... lines 38 - 66
}
```

The `getCredentials()` method is called on **every single request** and its job is either to fetch the username/password from the request and return them.

So, from here, there are 2 possibilities:

#	Conditions	Result	Next Step
A)	Return non-null value	Authentication continues	getUser()
B)	Return null	Authentication is skipped	Nothing! But if the user is anonymous and tries to access a secure page, getLoginUrl() will be called

A) If the URL is `/login_check` (that's the URL that our login form submits to), then we fetch the `_username` and `_password` post parameters (these were our form field names) and return them. Whatever you return here will be passed to a few other methods later. In this case - since we returned a non-null value from `getCredentials()` - the getUser() method is called next.

B) If the URL is *not* `/login_check`, we return `null`. In this case, the request continues anonymously - no other methods are called on the authenticator. If the page the user is accessing requires login, they'll be redirected to the login form: see getLoginUrl().

Tip

We also set a `Security::LAST_USERNAME` key into the session. This is optional, but it lets you pre-fill the login form with this value (see the SecurityController::loginAction from earlier).

getUser()

If `getCredentials()` returns a non-null value, then `getUser()` is called next. Its job is simple: return a user (an object implementing UserInterface):

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```
... lines 1 - 11
use Symfony\Component\Security\Core\User\UserProviderInterface;
... line 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 38
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['username'];

        return $userProvider->loadUserByUsername($username);
    }
... lines 45 - 66
}
```

The `$credentials` argument is whatever you returned from `getCredentials()` and the `$userProvider` is whatever you've configured in security.yml under the providers key. My provider queries the database and returns the `User` entity.

There are 2 paths from there:

#	Conditions	Result	Next Step
A)	Return a User object	Authentication continues	checkCredentials()
B)	Return null or throw an `AuthenticationException`	Authentication fails	Redirect to getLoginUrl()

A) If you return some `User` object (using whatever method you want) - then you'll continue on to checkCredentials().

B If you return `null` or throw any `Symfony\Component\Security\Core\Exception\AuthenticationException`, authentication will fail and the user will be redirected back to the login page: see getLoginUrl().

checkCredentials()

If you return a user from `getUser()`, then `checkCredentials()` is called next. Your job is simple: check if the username/password combination is valid. If it isn't, throw a `BadCredentialsException` (or any `AuthenticationException`):

```
68 lines | src/AppBundle/Security/FormLoginAuthenticator.php
... lines 1 - 8
use Symfony\Component\Security\Core\Exception\BadCredentialsException;
... line 10
use Symfony\Component\Security\Core\User\UserInterface;
... lines 12 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 45
    public function checkCredentials($credentials, UserInterface $user)
    {
        $plainPassword = $credentials['password'];
        $encoder = $this->container->get('security.password_encoder');
        if (!$encoder->isPasswordValid($user, $plainPassword)) {
            // throw any AuthenticationException
            throw new BadCredentialsException();
        }
    }
... lines 55 - 66
}
```

Like before, `$credentials` is whatever you returned from `getCredentials()`. And now, the `$user` argument is what you just returned from `getUser()`. To check the user, you can use the `security.password_encoder`, which automatically hashes the plain password based on your `security.yml` configuration.

Want to do some other custom checks beyond the password? Go crazy! Based on what you do, there are 2 paths:

#	Conditions	Result	Next Step
A)	do anything *except* throwing an `AuthenticationException`	Authentication successful	Redirect the user (may involve getDefaultSuccessRedirectUrl())
B)	Throw any type of `AuthenticationException`	Authentication fails	Redirect to getLoginUrl()

If you *don't* throw an exception, congratulations! You're user is now authenticated, and will be redirected somewhere...

getDefaultSuccessRedirectUrl()

Your user is now authenticated. Woot! But, where should we redirect them? The `AbstractFormLoginAuthenticator` class takes care of *most* of this automatically. If the user originally tried to access a protected page (e.g. `/admin`) but was redirected to the login page, then they'll now be redirected back to that URL (so, `/admin`).

But what if the user went to `/login` directly? In that case, you'll need to decide where they should go. How about the homepage?

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
// ... lines 16 - 61
    protected function getDefaultSuccessRedirectUrl()
    {
        return $this->container->get('router')
            ->generate('homepage');
    }
}
```

This fetches the `router` service and redirects to a `homepage` route (change this to a real route in your application). But note: this method is *only* called if there isn't some previous page that user should be redirected to.

getLoginUrl()

If authentication fails in `getUser()` or `checkCredentials()`, the user will be redirected back to the login page. In this method, you just need to tell Symfony where your login page lives:

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
// ... lines 16 - 55
    protected function getLoginUrl()
    {
        return $this->container->get('router')
            ->generate('security_login');
    }
// ... lines 61 - 66
}
```

In our case, the login page route name is `security_login`.

Customize!

Try it out! You should be able to login, see login errors, and control most of the process. So what else can we customize?

- How can I login by username *or* email (or any other weird way)?
- How can I customize the error messages?
- How can I control/hook into what happens when login fails?
- How can I control/hook into what happens on login success?
- How can I add a CSRF token?

Chapter 3: How to Authenticate via an API Token

Suppose you want to have an API where users authenticate by sending an `X-TOKEN` header. With Guard, this is one of the *easiest* things you can setup.

For this example, we have a `User` entity that as an `$apiToken` property that's auto-generated for every user when they register:

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

```
... lines 1 - 19
class User implements UserInterface
{
... lines 22 - 48
    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $apiToken;
... lines 53 - 133
}
```

But your setup can look however you want: an independent `ApiToken` entity that relates to your `User`, no `User` entity at all, or api tokens that are validated in some other way entirely.

Installing Guard

Read the short Installation chapter to make sure you've got the bundle installed and enabled.

Creating an Authenticator

In Guard, the whole authentication process - reading the `X-TOKEN` header value, validating it, returning an error response, etc - is handled in a single class called an "Authenticator". Your authenticator can be as crazy as you want, as long as it implements KnpU\Guard\GuardAuthenticatorInterface.

Most of the time, you can extend a convenience class called `AbstractGuardAuthenticator`. Create a new `ApiTokenAuthenticator` class, make it extend this class, and add all the necessary methods:

49 lines | src/AppBundle/Security/ApiTokenAuthenticator.php

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

use KnpU\Guard\AbstractGuardAuthenticator;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
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;

class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
    public function getCredentials(Request $request)
    {
        // TODO: Implement getCredentials() method.
    }

    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.
    }

    public function start(Request $request, AuthenticationException $authException = null)
    {
        // TODO: Implement start() method.
    }
}
```

Your mission: fill in each method. We'll get to that in a second.

But to fill on those methods, we'll need to query the database. Let's pass the Doctrine `EntityManager` into our authenticator:

```
78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php
```

```
... lines 1 - 4
use Doctrine\ORM\EntityManager;
... lines 6 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }
... lines 24 - 76
}
```

Registering your Authenticator

Before filling in the methods, let's tell Symfony about our fancy new authenticator. First, register it as a service:

14 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 10
    app.api_token_authenticator:
        class: AppBundle\Security\ApiTokenAuthenticator
        arguments: ["@doctrine.orm.entity_manager"]
```

Next, update your `security.yml` file to use the new service:

27 lines | app/config/security.yml

```
... lines 1 - 18
        main:
... line 20
            knpu_guard:
                authenticators:
                    - app.form_login_authenticator
                    - app.api_token_authenticator
                # by default, use the start() function from FormLoginAuthenticator
                entry_point: app.form_login_authenticator
```

Your firewall (called `main` here) can look however you want, as long as it has a `knpu_guard` section under it with an `authenticators` key that includes the service name that you setup a second ago (`app.api_token_authenticator` in my example).

Tip

The other authenticator - `app.form_login_authenticator` - is for my login form. If you don't need to *also* allow users to login via a form, then you can remove this. The `entry_point` option is only needed if you have multiple authenticators. See How can I use Multiple Authenticators?.

Filling in the Authenticator Methods

Your authenticator is now being used by Symfony. So let's fill in each method:

getCredentials()

78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
... lines 9 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 18 - 24
    public function getCredentials(Request $request)
    {
        return $request->headers->get('X-TOKEN');
    }
... lines 29 - 76
}
```

The `getCredentials()` method is called on **every single request** and its job is to fetch the API token and return it.

Well, that's pretty simple. From here, there are 3 possibilities:

#	Conditions	Result	Next Step
A)	Return non-null value	Authentication continues	getUser()
B)	Return null + endpoint requires auth	Auth skipped, 401 response	start()
C)	Return null+ endpoint does not require auth	Auth skipped, user is anon	nothing

A) The `X-TOKEN` header exists, so this returns a non-null value. In this case, getUser() is called next.

B) The `X-TOKEN` header is missing, so this returns `null`. But, your application *does* require authentication (e.g. via `access_control` or an `isGranted()` call). In this case, see start().

C) The `X-TOKEN` header is missing, so this returns `null`. But the user is accessing an endpoint that does *not* require authentication. In this case, the request continues anonymously - no other methods are called on the authenticator.

getUser()

If `getCredentials()` returns a non-null value, then `getUser()` is called next. Its job is simple: return a the user (an object implementing `UserInterface`):

```
78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php
```

```php
... lines 1 - 10
use Symfony\Component\Security\Core\Exception\AuthenticationCredentialsNotFoundException;
... lines 12 - 13
use Symfony\Component\Security\Core\User\UserProviderInterface;
... line 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 18 - 29
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $user = $this->em->getRepository('AppBundle:User')
            ->findOneBy(array('apiToken' => $credentials));

        // we could just return null, but this allows us to control the message a bit more
        if (!$user) {
            throw new AuthenticationCredentialsNotFoundException();
        }

        return $user;
    }
... lines 42 - 76
}
```

The `$credentials` argument is whatever you returned from `getCredentials()`, and the `$userProvider` is whatever you've configured in security.yml under the providers key.

You can choose to use your provider, or you can do something else entirely to load the user. In this case, we're doing a simple query on the `User` entity to see which User (if any) has this `apiToken` value.

From here, there are 2 possibilities:

#	Conditions	Result	Next Step
A)	Return a User	Authentication continues	checkCredentials()
B)	Return null or throw AuthenticationException	Authentication fails	onAuthenticationFailure()

A) If you successfully return a `User` object, then, checkCredentials() is called next.

B) If you return `null` or throw any `Symfony\Component\Security\Core\Exception\AuthenticationException`, authentication will fail and onAuthenticationFailure() is called next.

checkCredentials()

If you return a user from `getUser()`, then `checkCredentials()` is called next. Here, you can do any additional checks for the validity of the token - or anything else you can think of. In this example, we're doing nothing:

```
80 lines | src/AppBundle/Security/ApiTokenAuthenticator.php
```

```
... lines 1 - 12
use Symfony\Component\Security\Core\User\UserInterface;
... lines 14 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 18 - 42
    public function checkCredentials($credentials, UserInterface $user)
    {
        // the fact that they had a valid token that *was* attached to a user
        // means that their credentials are correct. So, there's nothing
        // additional (like a password) to check here.
        return;
    }
... lines 50 - 78
}
```

Like before, `$credentials` is whatever you returned from `getCredentials()`. And now, the `$user` argument is what you just returned from `getUser()`.

From here, there are 2 possibilities:

#	Conditions	Result	Next Step
A)	do anything *except* throwing an `AuthenticationException`	Authentication successful	onAuthenticationSuccess()
B)	Throw any type of `AuthenticationException`	Authentication fails	onAuthenticationFailure()

A) If you *don't* throw an exception, congrats! You're authenticated! In this case, onAuthenticationSuccess() is called next.

B) If you perform extra checks and throw any `Symfony\Component\Security\Core\Exception\AuthenticationException`, authentication will fail and onAuthenticationFailure() is called next.

onAuthenticationSuccess

Your user is authenticated! Amazing! At this point, in an API, you usually want to simply let the request continue like normal:

```
78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php
... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
... line 9
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
... lines 11 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 18 - 57
    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        // do nothing - let the request just continue!
        return;
    }
... lines 63 - 76
}
```

If you return `null` from this method: the request continues to process through Symfony like normal (only now, the request is authenticated).

Alternatively, you could return a `Response` object here. If you did, that `Response` would be returned to the client directly, without executing the controller for this request. For an API, that's probably not what you want.

onAuthenticationFailure

If you return `null` from `getUser()` or throw any `AuthenticationException` from `getUser()` or `checkCredentials()`, then you'll end up here. Your job is to create a `Response` that should be sent back to the user to tell them what went wrong:

78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php

```php
// ... lines 1 - 6
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
// ... lines 9 - 11
use Symfony\Component\Security\Core\Exception\AuthenticationException;
// ... lines 13 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
// ... lines 18 - 48
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        return new JsonResponse(
            // you could translate the message
            array('message' => $exception->getMessageKey()),
            403
        );
    }
// ... lines 57 - 76
}
```

In this case, we'll return a 403 Forbidden JSON response with a message about what went wrong. The `$exception` argument is the actual `AuthenticationException` that was thrown. It has a `getMessageKey()` method that contains a safe message about the authentication problem.

This Response will be sent back to the client - the controller will never be executed for this request.

start()

This method is called if an anomymous user accesses en endpoint that requires authentication. For our example, this would happen if the `X-TOKEN` header is empty (and so `getCredentials()`) returns `null`. Our job here is to return a Response that instructs the user that they need to re-send the request with authentication information (i.e. the `X-TOKEN` header):

78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php

```php
// ... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
// ... lines 9 - 11
use Symfony\Component\Security\Core\Exception\AuthenticationException;
// ... lines 13 - 68
    public function start(Request $request, AuthenticationException $authException = null)
    {
        return new JsonResponse(
            // you could translate the message
            array('message' => 'Authentication required'),
            401
        );
    }
}
```

In this case, we'll return a 401 Unauthorized JSON response.

supportsRememberMe

This method is required for all authenticators. If `true`, then the authenticator will work together with the `remember_me` functionality on your firewall, if you have it configured. Obviously, for an API, we don't need remember me functionality:

```
78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php
... lines 1 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 18 - 63
    public function supportsRememberMe()
    {
        return false;
    }
... lines 68 - 76
}
```

Testing your Endpoint

Yes! API token authentication is all setup. Let's test it with a simple script.

First, install Guzzle:

```
$ composer require guzzlehttp/guzzle:~6.0
```

Next, create a little "play" file at the root of your project that will make a request to our app. This assume your web server is running on `localhost:8000`:

```
18 lines | testAuth.php
... lines 1 - 2
require __DIR__.'/vendor/autoload.php';

$client = new GuzzleHttp\Client();
$res = $client->get('http://localhost:8000/secure', [
    'allow_redirects' => false,
    'http_errors' => false,
    'headers' => [
        // token for anna_admin in LoadUserData fixtures
        'X-Token' => 'ABCD1234'
    ]
]);

echo sprintf("Status Code: %s\n\n", $res->getStatusCode());
echo $res->getBody();
echo "\n\n";
```

In our app, the `anna_admin` user in the database has an `apiToken` of `ABCD1234`. In other words, this *should* work. Try it out from the command line:

```
$ php testAuth.php
```

If you see a 200 status code with response of "It works!"... well, um... it works! The `/secure` controller requires authentication. Now try changing the token value (or removing it entirely) to see our error responses.

Chapter 4: Social Login with Facebook

Everybody wants their site to have a "Login with Facebook", "Login with GitHub" or "Login with InstaFaceTweet". Let's give the people what they want!

Setting this up will take some coding, but the result will be easy to understand and simple to extend. Let's do it!

Tip

Watch our OAuth2 in 8 Steps tutorial first to get handle on how OAuth works.

The Flow

Social authentication uses OAuth - usually the authorization code grant type. That just means that we have a flow that looks like this:

TODO - IMAGE HERE

1. Your user clicks on a link to "Login with Facebook". This takes them to a Symfony controller on your site (e.g. `/connect/facebook`)
2. That controller redirects to Facebook, where they grant your application access
3. Facebook redirects back to your site (e.g. `/connect/facebook-check`) with a `?code=` query parameter
4. We make an API request back to Facebook to exchange this code for an access token. Then, immediately, we use this access token to make another API request to Facebook to fetch the user's information - like their email address. If we find an existing user, we can log the user in. If not, we might choose to create a User in the database, or have the user complete a "registration" form.

Installing Guard

Read the short Installation chapter to make sure you've got the bundle installed and enabled.

Creating your Facebook Application

To follow along, you'll need to create a Facebook Application at https://developers.facebook.com/. You can name it anything, but for the "Site URL", make sure it uses whatever domain you're using. In my case, I'm using `http://localhost:8000/`. I'll probably create a different application for my production site.

When you're done, this will give you "App ID" and "App Secret". Keep those handy!

Installing the OAuth Client Libraries

To help with the OAuth heavy-lifting, we'll use a nice oauth2-client library, and its oauth2-facebook helper library. Get these installed:

```
$ composer require league/oauth2-client:~1.0@dev league/oauth2-facebook
```

Setting up the Facebook Provider Service

The oauth2-facebook library lets us create a nice `Facebook` object that makes doing OAuth with Facebook a breeze. We'll need this object in several places, so let's register it as a service:

```
30 lines | app/config/services.yml
```

```
... lines 1 - 5
services:
... lines 7 - 14
    app.facebook_provider:
        class: League\OAuth2\Client\Provider\Facebook
        arguments:
            -
                clientId: %facebook_app_id%
                clientSecret: %facebook_app_secret%
                graphApiVersion: v2.3
                redirectUri: "@=service('router').generate('connect_facebook_check', {}, true)"
... lines 23 - 30
```

This references two new parameters - `facebook_app_id` and `facebook_app_secret`. Add these to your `parameters.yml` (and `parameters.yml.dist`) file with your Facebook application's "App ID" and "App Secret" values:

16 lines | app/config/parameters.yml

```
... line 1
parameters:
... lines 3 - 12
    facebook_app_id:  XXXX
    facebook_app_secret: XXXX
... lines 15 - 16
```

Tip

If you haven't seen the odd `"@=service('router')..."` expression syntax before, we have a blog post on it: Symfony Service Expressions: Do things you thought Impossible.

Creating the FacebookConnectController

Don't think about security yet! Instead, look at the flow above. We'll need a controller with *two* actions: one that simply redirects to Facebook for authorization (`/connect/facebook`) and another that handles what happens when Facebook redirects back to us (`/connect/facebook-check`):

37 lines | src/AppBundle/Controller/FacebookConnectController.php

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

class FacebookConnectController extends Controller
{
    /**
     * @Route("/connect/facebook", name="connect_facebook")
     */
    public function connectFacebookAction(Request $request)
    {
        // redirect to Facebook
        $facebookOAuthProvider = $this->get('app.facebook_provider');

        $url = $facebookOAuthProvider->getAuthorizationUrl([
            // these are actually the default scopes
            'scopes' => ['public_profile', 'email'],
        ]);

        return $this->redirect($url);
    }

    /**
     * @Route("/connect/facebook-check", name="connect_facebook_check")
     */
    public function connectFacebookActionCheck()
    {
        // will not be reached!
    }
}
```

The first URL - /connect/facebook - uses the Facebook provider service from the oauth2-client library that we just setup. Its job is simple: redirect to Facebook to start the authorization process. In a second, we'll add a "Login with Facebook" link that will point here.

The second URL - /connect/facebook-check - will be the URL that Facebook will redirect back to after. But notice it doesn't do anything - and it never will. Another layer (the authenticator) will intercept things and handle all the logic.

For good measure, let's create a "Login with Facebook" on our normal login page:

```
27 lines | app/Resources/views/security/login.html.twig
... lines 1 - 23
        <a href="{{ path('connect_facebook') }}">Login with Facebook</a>
... lines 25 - 27
```

Creating an Authenticator

With Guard, the whole authentication process - fetching the access token, getting the user information, redirecting after success, etc - is handled in a single class called an "Authenticator". Your authenticator can be as crazy as you want, as long as it implements KnpU\Guard\GuardAuthenticatorInterface.

Most of the time, you can extend a convenience class called AbstractGuardAuthenticator. Create a new FacebookAuthenticator class, make it extend this class, and add all the necessary methods:

49 lines | src/AppBundle/Security/FacebookAuthenticator.php

```php
... lines 1 - 2
namespace AppBundle\Security;

use KnpU\Guard\AbstractGuardAuthenticator;
use Symfony\Component\HttpFoundation\Request;
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;

class FacebookAuthenticator extends AbstractGuardAuthenticator
{
    public function getCredentials(Request $request)
    {
        // todo
    }

    public function getUser($authorizationCode, UserProviderInterface $userProvider)
    {
        // todo
    }

    public function checkCredentials($credentials, UserInterface $user)
    {
        // todo
    }

    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        // todo
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        // todo
    }

    public function supportsRememberMe()
    {
        // todo
    }

    public function start(Request $request, AuthenticationException $authException = null)
    {
        // todo
    }
}
```

Your mission: fill in each method. We'll get to that in a second.

To fill in those methods, we're going to need some services. To keep this tutorial simple, let's pass the entire container into our authenticator:

162 lines | src/AppBundle/Security/FacebookAuthenticator.php

```
... lines 1 - 10
use Symfony\Component\DependencyInjection\ContainerInterface;
... lines 12 - 22
class FacebookAuthenticator extends AbstractGuardAuthenticator
{
    private $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }
... lines 31 - 160
}
```

Tip

For seasoned-Symfony devs, you can of course inject *only* the services you need.

Registering your Authenticator

Before filling in the methods, let's tell Symfony about our fancy new authenticator. First, register it as a service:

30 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 23
    app.facebook_authenticator:
        class: AppBundle\Security\FacebookAuthenticator
        arguments:
            # you can also inject the services you need individually
            # except for the router, as it causes a circular reference problem
            - @service_container
```

Next, update your `security.yml` file to use the new service:

28 lines | app/config/security.yml

```
security:
... lines 2 - 13
    firewalls:
... lines 15 - 18
        main:
... line 20
            knpu_guard:
                authenticators:
... lines 23 - 24
                    - app.facebook_authenticator
... lines 26 - 28
```

Your firewall (called `main` here) can look however you want, as long as it has a `knpu_guard` section under it with an `authenticators` key that includes the service name that you setup a second ago (`app.facebook_authenticator` in my example).

Filling in the Authenticator Methods

Congratulations! Your authenticator is now being used by Symfony. Here's the flow so far:

1) The user clicks "Login with Facebook"; 2) Our `connectFacebookAction` redirects the user to Facebook; 3) After authorizing our app, Facebook redirects back to `/connect/facebook-connect`; 4) The `getCredentials()` method on `FacebookAuthenticator` is called and we start working our magic!

getCredentials()

162 lines | src/AppBundle/Security/FacebookAuthenticator.php

```
... lines 1 - 22
class FacebookAuthenticator extends AbstractGuardAuthenticator
{
... lines 25 - 31
    public function getCredentials(Request $request)
    {
        if ($request->getPathInfo() != '/connect/facebook-check') {
            // skip authentication unless we're on this URL!
            return null;
        }

        if ($code = $request->query->get('code')) {
            return $code;
        }

        // no code! Something went wrong. Quite probably the user denied our app access
        // you could read the error, error_code, error_description, error_reason query params
        // http://localhost:8000/connect/facebook-check?error=access_denied&error_code=200&error_description=Permissi
        throw CustomAuthenticationException::createWithSafeMessage(
            'There was an error getting access from Facebook. Please try again.'
        );
    }
... lines 50 - 160
}
```

If the user approves our application, Facebook will redirect back to `/connect/facebook-connect` with a `?code=ABC123` query parameter. That's called the "authorization code".

The `getCredentials()` method is called on **every single request** and its job is simple: grab this "authorization code" and return it.

Inside `getCredentials()`, here are 2 possible paths:

#	Conditions	Result	Next Step
A)	Return non-null value	Authentication continues	getUser()
B)	Throw an exception	Authentication fails	onAuthenticationFailure()
C)	Return null	Authentication is skipped	Nothing!

A) If the URL is `/connect/facebook-connect`, then we fetch the `code` query parameter that Facebook is sending us and return it. This will be passed to a few other methods later. In this case - since we returned a non-null value from `getCredentials()` - the getUser() method is called next.

B) If the URL is `/connect/facebook-connect` but there is no `code` query parameter, something went wrong! This probably means the user didn't authorize our app. To fail authentication, you can throw any `AuthenticationException`. The CustomAuthenticationException is just a cool way to control the message the user sees.

C) If the URL is *not* `/connect/facebook-connect`, we return `null`. In this case, the request continues anonymously - no other methods are called on the authenticator.

getUser()

If `getCredentials()` returns a non-null value, then `getUser()` is called next. Its job is simple: return a user (an object implementing UserInterface).

But to do that, there are several steps. Ultimately, there are two possible results:

#	Conditions	Result	Next Step
A)	Return a User object	Authentication continues	checkCredentials()
B)	Return null or throw an `AuthenticationException`	Authentication fails	Redirect to getLoginUrl()

getUser() Part 1: Get the access token

The `$authorizationCode` argument is whatever you returned from `getCredentials()`. Our first job is to talk to the Facebook API and exchange this for an "access token". Fortunately, with the oauth2-client library, this is easy:

```
162 lines | src/AppBundle/Security/FacebookAuthenticator.php
... lines 1 - 50
    public function getUser($authorizationCode, UserProviderInterface $userProvider)
    {
        $facebookProvider = $this->container->get('app.facebook_provider');

        try {
            // the credentials are really the access token
            $accessToken = $facebookProvider->getAccessToken(
                'authorization_code',
                ['code' => $authorizationCode]
            );
        } catch (IdentityProviderException $e) {
            // probably the authorization code has been used already
            $response = $e->getResponseBody();
            $errorCode = $response['error']['code'];
            $message = $response['error']['message'];

            throw CustomAuthenticationException::createWithSafeMessage(
                'There was an error logging you into Facebook - code '.$errorCode
            );
        }
... lines 71 - 105
    }
... lines 107 - 162
```

If this fails for some reason, we throw an AuthenticationException (specifically a `CustomAuthenticationException` to control the message).

getUser() Part 2: Get Facebook User Information

Now that we have a valid access token, we can make reqeusts to the Facebook API on behalf of the user. The most important thing we need is information about the user - like what is their email address?

To get that, use the `getResourceOwner()` method:

```
162 lines | src/AppBundle/Security/FacebookAuthenticator.php
... lines 1 - 50
    public function getUser($authorizationCode, UserProviderInterface $userProvider)
    {
... lines 53 - 71
        /** @var FacebookUser $facebookUser */
        $facebookUser = $facebookProvider->getResourceOwner($accessToken);
        $email = $facebookUser->getEmail();
... lines 75 - 105
    }
... lines 107 - 162
```

This returns a `FacebookUser` from the `oauth2-facebook` library (if you connect to something else like GitHub, it will have a different object). We can use that to get the user's email address.

getUser() Part 3: Fetching/Creating the User

Great! We now know some information about the user, including their email address.

A) If you return some `User` object (using whatever method you want) - then you'll continue on to checkCredentials().

B If you return `null` or throw any `Symfony\Component\Security\Core\Exception\AuthenticationException`, authentication will fail and the user will be redirected back to the login page: see getLoginUrl().

checkCredentials()

If you return a user from `getUser()`, then `checkCredentials()` is called next. Your job is simple: check if the username/password combination is valid. If it isn't, throw a `BadCredentialsException` (or any `AuthenticationException`):

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```
... lines 1 - 8
use Symfony\Component\Security\Core\Exception\BadCredentialsException;
... line 10
use Symfony\Component\Security\Core\User\UserInterface;
... lines 12 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 45
    public function checkCredentials($credentials, UserInterface $user)
    {
        $plainPassword = $credentials['password'];
        $encoder = $this->container->get('security.password_encoder');
        if (!$encoder->isPasswordValid($user, $plainPassword)) {
            // throw any AuthenticationException
            throw new BadCredentialsException();
        }
    }
... lines 55 - 66
}
```

Like before, `$credentials` is whatever you returned from `getCredentials()`. And now, the `$user` argument is what you just returned from `getUser()`. To check the user, you can use the `security.password_encoder`, which automatically hashes the plain password based on your `security.yml` configuration.

Want to do some other custom checks beyond the password? Go crazy! Based on what you do, there are 2 paths:

#	Conditions	Result	Next Step
A)	do anything *except* throwing an `AuthenticationException`	Authentication successful	Redirect the user (may involve getDefaultSuccessRedirectUrl())
B)	Throw any type of `AuthenticationException`	Authentication fails	Redirect to getLoginUrl()

If you *don't* throw an exception, congratulations! You're user is now authenticated, and will be redirected somewhere...

getDefaultSuccessRedirectUrl()

Your user is now authenticated. Woot! But, where should we redirect them? The `AbstractFormLoginAuthenticator` class takes care of *most* of this automatically. If the user originally tried to access a protected page (e.g. `/admin`) but was redirected to the login page, then they'll now be redirected back to that URL (so, `/admin`).

But what if the user went to `/login` directly? In that case, you'll need to decide where they should go. How about the homepage?

```
68 lines | src/AppBundle/Security/FormLoginAuthenticator.php
... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 61
    protected function getDefaultSuccessRedirectUrl()
    {
        return $this->container->get('router')
            ->generate('homepage');
    }
}
```

This fetches the `router` service and redirects to a `homepage` route (change this to a real route in your application). But note: this method is *only* called if there isn't some previous page that user should be redirected to.

getLoginUrl()

If authentication fails in `getUser()` or `checkCredentials()`, the user will be redirected back to the login page. In this method, you just need to tell Symfony where your login page lives:

```
68 lines | src/AppBundle/Security/FormLoginAuthenticator.php
... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 55
    protected function getLoginUrl()
    {
        return $this->container->get('router')
            ->generate('security_login');
    }
... lines 61 - 66
}
```

In our case, the login page route name is `security_login`.

Installing Guard

Read the short Installation chapter to make sure you've got the bundle installed and enabled.

Creating an Authenticator

With Guard, the whole authentication process - fetching the username/password POST values, validating the password, redirecting after success, etc - is handled in a single class called an "Authenticator". Your authenticator can be as crazy as you want, as long as it implements KnpU\Guard\GuardAuthenticatorInterface.

For login forms, life is easier, thanks to a convenience class called `AbstractFormLoginAuthenticator`. Create a new `FormLoginAuthenticator` class, make it extend this class, and add all the missing methods (from the interface and abstract class):

```
37 lines | src/AppBundle/Security/FormLoginAuthenticator.php
```

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

use KnpU\Guard\Authenticator\AbstractFormLoginAuthenticator;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;

class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
    public function getCredentials(Request $request)
    {
        // TODO: Implement getCredentials() method.
    }

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

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

    protected function getLoginUrl()
    {
        // TODO: Implement getLoginUrl() method.
    }

    protected function getDefaultSuccessRedirectUrl()
    {
        // TODO: Implement getDefaultSuccessRedirectUrl() method.
    }
}
```

Your mission: fill in each method. We'll get to that in a second.

To fill in those methods, we're going to need some services. To keep this tutorial simple, let's pass the entire container into our authenticator:

45 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```
... lines 1 - 5
use Symfony\Component\DependencyInjection\ContainerInterface;
... lines 7 - 10
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
    private $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }
... lines 19 - 43
}
```

Tip

For seasoned-Symfony devs, you can of course inject *only* the services you need.

Registering your Authenticator

Before filling in the methods, let's tell Symfony about our fancy new authenticator. First, register it as a service:

10 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
    app.form_login_authenticator:
        class: AppBundle\Security\FormLoginAuthenticator
        arguments: ["@service_container"]
```

Next, update your `security.yml` file to use the new service:

24 lines | app/config/security.yml

```yaml
security:
... lines 2 - 13
    firewalls:
... lines 15 - 18
        main:
            anonymous: ~
            knpu_guard:
                authenticators:
                    - app.form_login_authenticator
```

Your firewall (called `main` here) can look however you want, as long as it has a `knpu_guard` section under it with an `authenticators` key that includes the service name that you setup a second ago (`app.form_login_authenticator` in my example).

I've also setup my "user provider" to load my users from the database:

24 lines | app/config/security.yml

```yaml
security:

    encoders:
        # Our user class and the algorithm we'll use to encode passwords
        # http://symfony.com/doc/current/book/security.html#encoding-the-user-s-password
        AppBundle\Entity\User: bcrypt

    providers:
        # Simple example of loading users via Doctrine
        # To load users from somewhere else: http://symfony.com/doc/current/cookbook/security/custom_provider.html
        database_users:
            entity: { class: AppBundle:User, property: username }
... lines 13 - 24
```

In a minute, you'll see where that's used.

Filling in the Authenticator Methods

Your authenticator is now being used by Symfony. So let's fill in each method:

getCredentials()

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```
... lines 1 - 6
use Symfony\Component\HttpFoundation\Request;
... lines 8 - 9
use Symfony\Component\Security\Core\Security;
... lines 11 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 22
    public function getCredentials(Request $request)
    {
        if ($request->getPathInfo() != '/login_check') {
            return;
        }

        $username = $request->request->get('_username');
        $request->getSession()->set(Security::LAST_USERNAME, $username);
        $password = $request->request->get('_password');

        return array(
            'username' => $username,
            'password' => $password
        );
    }
... lines 38 - 66
}
```

The `getCredentials()` method is called on **every single request** and its job is either to fetch the username/password from the request and return them.

So, from here, there are 2 possibilities:

#	Conditions	Result	Next Step
A)	Return non-null value	Authentication continues	getUser()
B)	Return null	Authentication is skipped	Nothing! But if the user is anonymous and tries to access a secure page, getLoginUrl() will be called

A) If the URL is `/login_check` (that's the URL that our login form submits to), then we fetch the `_username` and `_password` post parameters (these were our form field names) and return them. Whatever you return here will be passed to a few other methods later. In this case - since we returned a non-null value from `getCredentials()` - the getUser() method is called next.

B) If the URL is *not* `/login_check`, we return `null`. In this case, the request continues anonymously - no other methods are called on the authenticator. If the page the user is accessing requires login, they'll be redirected to the login form: see getLoginUrl().

Tip

We also set a `Security::LAST_USERNAME` key into the session. This is optional, but it lets you pre-fill the login form with this value (see the SecurityController::loginAction from earlier).

getUser()

If `getCredentials()` returns a non-null value, then `getUser()` is called next. Its job is simple: return a user (an object implementing UserInterface):

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
... lines 1 - 11
use Symfony\Component\Security\Core\User\UserProviderInterface;
... line 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 38
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['username'];

        return $userProvider->loadUserByUsername($username);
    }
... lines 45 - 66
}
```

The `$credentials` argument is whatever you returned from `getCredentials()` and the `$userProvider` is whatever you've configured in security.yml under the providers key. My provider queries the database and returns the `User` entity.

There are 2 paths from there:

#	Conditions	Result	Next Step
A)	Return a User object	Authentication continues	checkCredentials()
B)	Return null or throw an `AuthenticationException`	Authentication fails	Redirect to getLoginUrl()

A) If you return some `User` object (using whatever method you want) - then you'll continue on to checkCredentials().

B If you return `null` or throw any `Symfony\Component\Security\Core\Exception\AuthenticationException`, authentication will fail and the user will be redirected back to the login page: see getLoginUrl().

checkCredentials()

If you return a user from `getUser()`, then `checkCredentials()` is called next. Your job is simple: check if the username/password combination is valid. If it isn't, throw a `BadCredentialsException` (or any `AuthenticationException`):

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
... lines 1 - 8
use Symfony\Component\Security\Core\Exception\BadCredentialsException;
... line 10
use Symfony\Component\Security\Core\User\UserInterface;
... lines 12 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 45
    public function checkCredentials($credentials, UserInterface $user)
    {
        $plainPassword = $credentials['password'];
        $encoder = $this->container->get('security.password_encoder');
        if (!$encoder->isPasswordValid($user, $plainPassword)) {
            // throw any AuthenticationException
            throw new BadCredentialsException();
        }
    }
... lines 55 - 66
}
```

Like before, `$credentials` is whatever you returned from `getCredentials()`. And now, the `$user` argument is what you just returned from `getUser()`. To check the user, you can use the `security.password_encoder`, which automatically hashes the plain password based on your `security.yml` configuration.

Want to do some other custom checks beyond the password? Go crazy! Based on what you do, there are 2 paths:

#	Conditions	Result	Next Step
A)	do anything *except* throwing an `AuthenticationException`	Authentication successful	Redirect the user (may involve getDefaultSuccessRedirectUrl())
B)	Throw any type of `AuthenticationException`	Authentication fails	Redirect to getLoginUrl()

If you *don't* throw an exception, congratulations! You're user is now authenticated, and will be redirected somewhere...

getDefaultSuccessRedirectUrl()

Your user is now authenticated. Woot! But, where should we redirect them? The `AbstractFormLoginAuthenticator` class takes care of *most* of this automatically. If the user originally tried to access a protected page (e.g. `/admin`) but was redirected to the login page, then they'll now be redirected back to that URL (so, `/admin`).

But what if the user went to `/login` directly? In that case, you'll need to decide where they should go. How about the homepage?

```
68 lines | src/AppBundle/Security/FormLoginAuthenticator.php
... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 61
    protected function getDefaultSuccessRedirectUrl()
    {
        return $this->container->get('router')
            ->generate('homepage');
    }
}
```

This fetches the `router` service and redirects to a `homepage` route (change this to a real route in your application). But note: this method is *only* called if there isn't some previous page that user should be redirected to.

getLoginUrl()

If authentication fails in `getUser()` or `checkCredentials()`, the user will be redirected back to the login page. In this method, you just need to tell Symfony where your login page lives:

```
68 lines | src/AppBundle/Security/FormLoginAuthenticator.php
... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 55
    protected function getLoginUrl()
    {
        return $this->container->get('router')
            ->generate('security_login');
    }
... lines 61 - 66
}
```

In our case, the login page route name is `security_login`.

Customize!

Try it out! You should be able to login, see login errors, and control most of the process. So what else can we customize?

- How can I login by username *or* email (or any other weird way)?
- How can I customize the error messages?
- How can I control/hook into what happens when login fails?
- How can I control/hook into what happens on login success?
- How can I add a CSRF token?

Chapter 5: How to Login with a username *or* email (or crazier)

Whenever you login, you identify yourself. For a form, this might be with a username or email. With an API, the token often serves both to identify *who* you are and serve as a sort of "password".

With Guard, you can use any crazy combination of methods to figure out *who* is trying to authenticate. The only rule is that your getUser function returns *some* object that implements UserInterface.

Let's look at an example of *how* you could customize this:

Logging in with a username *or* email

In the Form Login chapter, we built a login form that queries for a user from the database using the `username` property:

68 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 11
use Symfony\Component\Security\Core\User\UserProviderInterface;
// ... line 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
// ... lines 16 - 38
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['username'];

        return $userProvider->loadUserByUsername($username);
    }
// ... lines 45 - 66
}
```

But what if we wanted to let the user enter his username *or* email? First, create a method inside your `UserRepository` for this query:

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

```php
// ... lines 1 - 25
class UserRepository extends EntityRepository
{
    /**
     * @param string $username
     * @return User
     */
    public function findByUsernameOrEmail($username)
    {
        return $this->createQueryBuilder('u')
            ->andWhere('u.username = :username OR u.email = :username')
            ->setParameter('username', $username)
            ->getQuery()
            ->getOneOrNullResult();
    }
}
```

Now, in `getUser()`, simply call this method to return your `User` object:

72 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```
... lines 1 - 13
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 38
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['username'];
        $userRepo = $this->container
            ->get('doctrine')
            ->getManager()
            ->getRepository('AppBundle:User');

        return $userRepo->findByUsernameOrEmail($username);
    }
... lines 49 - 70
}
```

This works because we're injecting the entire service container. But, you could just as easily inject *only* the entity manager to clean things up.

Now, wasn't that easy? Have some other weird requirement for how a user is loaded? Do whatever you want inside of `getUser()`.

Tip

Why not use the `$userProvider` argument? The `$userProvider` that's passed to us here is what we have configured in `security.yml` under the providers key. In this project, this object gives us a `loadUserByUsername` method that queries for the `User` by the username. We *could* customize the user provider and make it do what we want. Or, we could simply fetch our repository directly and query for what we need. That seems much easier, and I've yet to see a downside.

Chapter 6: Customizing Authentication Failure Messages

Authentication can fail for a lot of reasons: an invalid username, bad password, locked account, etc, etc. And whether we're building a login form or an API, you need to give your users the *best* possible error message so they know how to fix things. If your error message is "Authentication error" when they type in a bad password, you're doing it wrong.

How and Where to Fail Authentication

Authentication can fail inside your authenticator in any of these 3 functions:

- `getCredentials()`
- `getUser()`
- `checkCredentials()`

Causing an authentication failure is easy: simply throw *any* instance of Symfony's `Symfony\Component\Security\Core\Exception\AuthenticationException`. In fact, if you return `null` from `getUser()`, Guard automatically throws a UsernameNotFoundException, which extends `AuthenticationException`.

Controlling the Message with CustomAuthenticationException

Any class that extends `AuthenticationException` has a hardcoded message that it causes. Here are some examples:

Class	**Message**
UsernameNotFoundException	`Username could not be found.`
BadCredentialsException	`Invalid credentials.`
AccountExpiredException	`Account has expired.`

Unfortunately, you *cannot* change these messages dynamically. In normal Symfony, you either need to translate these message or create a *new* exception class that extends `AuthenticationException` and customize your message there.

But wait! Guard comes with a class to help: CustomAuthenticationException. Use it inside any of the 3 methods above to customize your error message:

```
81 lines | src/AppBundle/Security/FormLoginAuthenticator.php
```

```
... lines 1 - 14
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 17 - 39
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['username'];

        // a silly example of failing with a custom message
        if ($username == 'rails_troll') {
            throw CustomAuthenticationException::createWithSafeMessage(
                'Get outta here rails_troll - we don\'t like you!'
            );
        }
... lines 50 - 56
    }
... lines 58 - 79
}
```

Using the Message in onAuthenticationFailure

Whenever any type of `AuthenticationException` is thrown in the process, the `onAuthenticationFailure()` method is called on your authenticator. Its second argument - `$exception` - will be this exception. Use its `getMessageKey()` to fetch the correct message:

78 lines | src/AppBundle/Security/ApiTokenAuthenticator.php

```
... lines 1 - 15
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 18 - 48
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        return new JsonResponse(
            // you could translate the message
            array('message' => $exception->getMessageKey()),
            403
        );
    }
... lines 57 - 76
}
```

Tip

if you're using the `AbstractFormLoginAuthenticator` base class, the `onAuthenticationFailure()` method is taken care of for you, but you can override it if you need to.

Of course, you can really use whatever logic you want in here to return a nice message to the user.

Have fun and give friendly errors!

Chapter 7: Customizing Failure Handling

Authentication can fail inside your authenticator in any of these 3 functions:

- `getCredentials()`
- `getUser()`
- `checkCredentials()`

The Customizing Authentication Failure Messages tutorial tells you *how* you can fail authentication and how to customize the error message when that happens.

But if you need more control, use the `onAuthenticationFailure()` method.

onAuthenticationFailure()

Every authenticator has a `onAuthenticationFailure()` method. This is called whenever authentication fails, and it has one job: create a `Response` that should be sent back to the user. This could be a redirect back to the login page or a 403 JSON response.

If you extend certain authenticators - like `AbstractFormLoginAuthenticator` - then this method is filled in for you automatically. But you can feel free to override it and customize.

Sending back JSON for AJAX

Suppose your login form uses AJAX. Instead of redirecting to `/login` on a failure, you probably want it to return some sort of JSON response. Just override `onAuthenticationFailure()` :

98 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 7
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
// ... line 10
use Symfony\Component\Security\Core\Exception\AuthenticationException;
// ... lines 12 - 16
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
// ... lines 19 - 70
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        // AJAX! Maybe return some JSON
        if ($request->isXmlHttpRequest()) {
            return new JsonResponse(
                // you could translate the message
                array('message' => $exception->getMessageKey()),
                403
            );
        }

        // for non-AJAX requests, return the normal redirect
        return parent::onAuthenticationFailure($request, $exception);
    }
// ... lines 85 - 96
}
```

That's it! If you fail authentication via AJAX, you'll receive a JSON response instead of the redirect.

Chapter 8: Customizing Success Handling

So, your authentication is working. Yes! Now, what if you need to hook into what happens next? For example, maybe you need to redirect to a special page, or return JSON instead of a redirect in some cases. Or perhaps you want to store the last login time of your user. All that is possible and easy.

onAuthenticationSuccess()

Every authenticator has a `onAuthenticationSuccess()` method. This is called whenever authentication is completed, and it has one job: create a `Response` that should be sent back to the user. This could be a redirect back to the last page the user visited or return `null` and let the request continue (see API token).

If you extend certain authenticators - like `AbstractFormLoginAuthenticator` - then this method is filled in for you automatically. But you can feel free to override it and customize.

Sending back JSON for AJAX

Suppose your login form uses AJAX. Instead of redirecting after success, you probably want it to return some sort of JSON response. Just override `onAuthenticationSuccess()`:

113 lines | src/AppBundle/Security/FormLoginAuthenticator.php

```php
// ... lines 1 - 7
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
// ... line 10
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
// ... lines 12 - 17
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
// ... lines 20 - 86
    public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
    {
        // AJAX! Return some JSON
        if ($request->isXmlHttpRequest()) {
            return new JsonResponse(
                // maybe send back the user's id
                array('userId' => $token->getUser()->getId())
            );
        }

        // for non-AJAX requests, return the normal redirect
        return parent::onAuthenticationSuccess($request, $token, $providerKey);
    }
// ... lines 100 - 111
}
```

That's it! If you login via AJAX, you'll receive a JSON response instead of the redirect.

Performing an Action on Login

Suppose you want to store the "last login" time for the user in the database. You *could* override `onAuthenticationSuccess()`, update the User and save.

But, there's a better way: Symfony security system dispatches a `security.interactive_login` event that you can hook into. Why is this better? Because this will be called whenever a user logs in, whether it is via this authenticator, another authenticator or some non-Guard system.

Tip

Everything in this section works in normal Symfony, even without Guard!

First, make sure you have a column on your user:

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

```
... lines 1 - 19
class User implements UserInterface
{
... lines 22 - 53
    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $lastLoginTime;
... lines 58 - 143

    public function setLastLoginTime(\DateTime $lastLoginTime)
    {
        $this->lastLoginTime = $lastLoginTime;
    }
}
```

Next, create an event subscriber. This will be called whenever a user logs in. It's job is simple: update this `lastLoginTime` property and save the User:

34 lines | src/AppBundle/EventListener/LastLoginSubscriber.php

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

use AppBundle\Entity\User;
use Doctrine\ORM\EntityManager;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Security\Http\Event\InteractiveLoginEvent;
use Symfony\Component\Security\Http\SecurityEvents;

class LastLoginSubscriber implements EventSubscriberInterface
{
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }

    public function onInteractiveLogin(InteractiveLoginEvent $event)
    {
        /** @var User $user */
        $user = $event->getAuthenticationToken()->getUser();
        $user->setLastLoginTime(new \DateTime());
        $this->em->persist($user);
        $this->em->flush($user);
    }

    public static function getSubscribedEvents()
    {
        return array(SecurityEvents::INTERACTIVE_LOGIN => 'onInteractiveLogin');
    }
}
```

Tip

Not familiar with listeners or susbcribers? Check out Interrupt Symfony with an Event Subscriber

Now, just register this as a service and tag it so that Symfony know about the subscriber:

20 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 14
    app.last_login_subscriber:
        class: AppBundle\EventListener\LastLoginSubscriber
        arguments: ["@doctrine.orm.entity_manager"]
        tags:
            - { name: kernel.event_subscriber }
```

That's all you need. Next time you login, the User's `lastLoginTime` will automatically be updated in the database.

Chapter 9: Login Form: Adding a CSRF Token

Come back soon - still in progress!

Chapter 10: How can I use Multiple Authenticators?

Todo - come back later!

Chapter 11: How can I use this in Silex?

Todo! Check back...

Chapter 12: FOSUserBundle

Todo - come back later!

Chapter 13: How can I Manually Authenticate a User?

In progress - check back later!