Symfony Security Voters (free cookies!)

With <3 from SymfonyCasts

Chapter 1: Symfony Security Voters (free cookies!)

SYMFONY SECURITY VOTERS (FREE COOKIES!)¶

See also

Voters have been updated in Symfony 2.8! Check out our updated tutorial about them: The new Voter Class.

Hey guys! It's getting a little colder in Michigan, Leanna and I are doing a little bit of baking, and baking makes me think of security. Specifically the kind of security that says: "no you can't eat my cookie because I baked it". And it actually has use inside of our applications because a lot of times we need to figure out whether the current user has access to edit, delete or view something. It's because of this we've cooked up a delicious little application which is going to show you one of my favorite and most underutilized features in Symfony: security voters.

Today's Application: DeliciousCookie¶

So I'll login using username `ryan` password `cookie` and basically we only have one page in this app which shows us all of these cookies that Ryan and Leanna have baked. Each of those has a "nom" button which allows me to eat the cookie, shows me a nice message and deletes it from the database. Really high-tech stuff.

The application is pretty straight forward: we have an `AppBundle` of course and inside of there we have a single entity called `DeliciousCookie`. The most important thing about `DeliciousCookie` is that there's a `bakerUsername` property which stores who actually baked this cookie. To keep this application simple I don't have a `User` entity so I'm just using a username string there:

```
// src/AppBundle/Entity/DeliciousCookie.php
// ...

/** @ORM\Entity */
class DeliciousCookie
{
    // ...

    /**
     * @ORM\Column()
     */
    private $flavor;

    /**
     * @ORM\Column(name="baker_username")
     */
    private $bakerUsername;

    // ...
}
```

Right now, anybody can eat any cookie no matter who baked it. But our goal is to make it so that you can only eat cookies that you've baked. The `CookieController` holds the page that actually lists the cookies and then there's one POST-only endpoint which handles actually deleting the cookie in the database and setting that nice flash message:

```php
// src/AppBundle/Controller/CookieController.php
// ...

class CookieController extends Controller
{
    /** @Route("/cookies", name="cookie_list") */
    public function indexAction()
    {
        $cookies = $this->getDoctrine()
            ->getRepository('AppBundle:DeliciousCookie')
            ->findAll();

        return $this->render('Cookie/index.html.twig', array(
            'cookies' => $cookies,
        ));
    }

    /**
     * @Route("/cookies/nom/{id}", name="cookie_nom")
     * @Method("POST")
     */
    public function nomAction(Request $request, $id)
    {
        $em = $this->getDoctrine()->getManager();
        $cookie = $em->getRepository('AppBundle:DeliciousCookie')
            ->find($id);

        // ...

        $em->remove($cookie);
        $em->flush();

        // some flash-setting stuff...

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

The only other interesting thing is `security.yml`. We have two hard-coded users: `ryan` and `leanna`, and I also have an `access_control` which requires login for everything under `/cookies`, which is why we had to login before we saw our cookie list. We take cookie security very seriously:

```yaml
# app/config/security.yml
security:
    # ...

    providers:
        in_memory:
            memory:
                users:
                    ryan:  { password: cookie, roles: 'ROLE_COOKIE_ENJOYER' }
                    leanna: { password: cookie, roles: 'ROLE_COOKIE_MONSTER' }


    firewalls:
        default:
            pattern: ^/
            anonymous: ~
            form_login: ~
            logout: ~

    access_control:
        - { path: ^/cookies, roles: IS_AUTHENTICATED_FULLY }
```

Preventing Access: The Easy Way¶

Preventing me from eating a cookie baked by someone else is actually pretty simple. And what we should do first is just put the logic into our controller. So I'll do that here: if the baker's username does not equal the current user's username, we're going to throw that `AccessDeniedException` and say: "Hey you didn't bake this!":

```
// src/AppBundle/Controller/CookieController.php
// ...

public function nomAction(Request $request, $id)
{
    $em = $this->getDoctrine()->getManager();
    $cookie = $em->getRepository('AppBundle:DeliciousCookie')
        ->find($id);

    // ...

    if ($cookie->getBakerUsername() != $this->getUser()->getUsername()) {
        throw $this->createAccessDeniedException(
            'You did not bake this delicious cookie!'
        );
    }
    // ...
}
```

Now if we try to eat one of Leanna's cookies she catches us with a nice clear message. And of course in the production environment, this would be your 403 error page.

Tip

See Error Pages for how to customize your 404, 403 and 500 error pages in production.

So what's the problem with this? The problem is that we also need to go into our Twig template and repeat the logic there:

```
{# app/Resources/views/Cookie/index.html.twig #}
{# ... #}

{% for cookie in cookies %}
    {# ... #}

    {% if cookie.bakerUsername == app.user.username %}
        <form action="{{ path('cookie_nom', {'id': cookie.id}) }}" method="POST">
            <button type="submit" class="btn">NOM! <i class="glyphicon glyphicon-cutlery"></i></button>
        </form>
    {% endif %}

    {# ... #}
{% endfor %}
```

And when it comes to security logic, especially security logic that protects cookies, we don't want to repeat it across your application. If you change something later and forget to update part of your security, you're going to have a big problem. But for now, I'll do it manually and we can see that the nom button hides or shows based on which cookies I actually baked.

Creating a Voter¶

So the goal of a voter is to allow us to centralize that logic so we only have it in one spot. I'll create a `Security` directory which is purely for organization and then put a `CookieVoter` inside of it. I'm using Symfony 2.6 for this project, which comes with a fantastic new AbstractVoter class which I'm going to use. If you're using Symfony 2.5 or lower, you can actually find this class on the internet and just use it in your project today. Just update the namespace to match your project and then extend it. This class doesn't have any external dependencies so it's going to work just fine in whatever Symfony version you have.

So I'll extend it and then use a really nice feature in PHPstorm to tell me the three abstract methods that I need to fill in:

```php
// src/AppBundle/Security/CookieVoter.php
namespace AppBundle\Security;

use Symfony\Component\Security\Core\Authorization\Voter\AbstractVoter;
use Symfony\Component\Security\Core\User\UserInterface;

class CookieVoter extends AbstractVoter
{
    protected function getSupportedClasses()
    {
        // todo
    }

    protected function getSupportedAttributes()
    {
        // todo
    }

    protected function isGranted($attribute, $object, $user = null)
    {
        // todo
    }
}
```

But What does a Voter Do?¶

So let me back up because I haven't actually told you what these voters do. First let me show you how I want our code to look when we're finished. Instead of doing the logic manually I'm going to use the `isGranted` function, pass it a string: `NOM` which is something I'm making up – you'll see why it's important in a second – and then pass the `$cookie` object as the second argument to `isGranted`:

```php
// src/AppBundle/Controller/CookieController.php
// ...

public function nomAction(Request $request, $id)
{
    $em = $this->getDoctrine()->getManager();
    $cookie = $em->getRepository('AppBundle:DeliciousCookie')
        ->find($id);

    // ...

    if (!$this->isGranted('NOM', $cookie)) {
        throw $this->createAccessDeniedException(
            'You did not bake this delicious cookie!'
        );
    }
    // ...
}
```

The `isGranted` shortcut is new to 2.6 but all it does is go out to the `security.context` service and call `isGranted` on it. So this is exactly what you're using in earlier projects. If you don't have the shortcut method just go out to the `security.context` service manually.

Behind the scenes, whenever you use the `isGranted` function, Symfony calls out to a bunch of voters and asks each of them if they can figure out whether or not we should have access. For example, whenever you pass `ROLE_SOMETHING` to `` `isGranted` `` like `` `ROLE_USER ``, there's a `RoleVoter`` class which tries to figure out whether the current user has whatever role you're asking about.

What most people don't realize is that you can invent these strings. So in this case I'm just inventing `NOM` and we're going to add a new voter into that system that says: "Hey Symfony! Whenever the `NOM` attribute is passed to `isGranted`, call me!" To get that to work we just need to fill in the `getSupportedClasses` and the `getSupportedAttributes` functions.

Filling in CookieVoter¶

First, in `getSupportedClasses`, were going to return the `DeliciousCookie` class string:

```
// src/AppBundle/Security/CookieVoter.php
// ...

protected function getSupportedClasses()
{
    return array('AppBundle\Entity\DeliciousCookie');
}
```

This tells Symfony that when we pass a `DeliciousCookie` object to `isGranted`, our voter should be called. We'll do the same thing in `getSupportedAttributes` and we'll return an array with the `NOM` string:

```
// src/AppBundle/Security/CookieVoter.php
// ...

protected function getSupportedAttributes()
{
    return array('NOM');
}
```

This tells Symfony that when we pass `NOM` to `isGranted` that *our* voter should be called. Whenever both of these functions return `true`, the `` `isGranted` `` function at the bottom of this class is going to be called.

For now I'll just use the glorious `var_dump` to print the attribute object and user and I'm going to put a die after that:

```
// src/AppBundle/Security/CookieVoter.php
// ...

protected function isGranted($attribute, $object, $user = null)
{
    var_dump($attribute, $object, $user);die;
}
```

Registering and Tagging your Voter¶

At this point, other than the crazy debug code I have at the bottom, our voter class is ready to go. But Symfony is not going to automatically find it. To tell Symfony about our new voter we're going to need to register it as a service and give it a special tag.

I have an `app/config/services.yml` file which I'm importing from my `config.yml` file, so we'll put the service there:

```
# app/config/services.yml
services:
    app_cookie_voter:
        class: AppBundle\Security\CookieVoter
        tags:
            - { name: security.voter }
```

The name doesn't matter but try to keep it relatively short. And the autocompleting I'm getting is from the nice Symfony2 plugin for PHPStorm. Our class doesn't have any constructor arguments yet so I'm just leaving that key off.

The really important part is `tags`. You need to add one tag whose name is `security.voter`. This is like raising our hand for our voter and saying: "Hey Symfony, whenever somebody calls `isGranted` I want *our* voter to actually be called."

So we have the voter, we have the service with the tag so let's try it out! When we refresh... Bam! We see things dumped out: proof that our voter is being called.

Adding your Cookie-Protecting Biz Logic¶

Now here's where things get really really cool. Now in theory because of my `access_control`, this voter should never be called unless the user is logged in. But just in case it is let's use `is_object` to check to see if the user is actually logged in:

```
// src/AppBundle/Security/CookieVoter.php
// ...

protected function isGranted($attribute, $object, $user = null)
{
    if (!is_object($user)) {
        return false;
    }

    // ... todo
}
```

Remember we need to do this because in Symfony 2 if you're anonymous the user is actually set to a string. From here it's pure business logic: if the Baker's username equals the user's username let's give them access. Otherwise let's not give them access:

```
// src/AppBundle/Security/CookieVoter.php
// ...

protected function isGranted($attribute, $object, $user = null)
{
    if (!is_object($user)) {
        return false;
    }

    if ($object->getBakerUsername() == $user->getUsername()) {
        return true;
    }

    return false;
}
```

So let's refresh the "nom" request ... and it works! We're logged in as Ryan and we are actually nomming a Ryan cookie so this make sense. Remember the goal of this was to centralize our logic. So now we can go into our Twig template and do the exact same thing there. We'll use the `is_granted` function, pass it nom and pass it the `cookie` object:

```
{# app/Resources/views/Cookie/index.html.twig #}
{# ... #}

{% for cookie in cookies %}
    {# ... #}

    {% if is_granted('NOM', cookie) %}
        <form action="{{ path('cookie_nom', {'id': cookie.id}) }}" method="POST">
            <button type="submit" class="btn">NOM! <i class="glyphicon glyphicon-cutlery"></i></button>
        </form>
    {% endif %}

    {# ... #}
{% endfor %}
```

And as you might expect, when we refresh, we see the exact same results as before except everything is pulling from that central voter.

Giving a `ROLE_COOKIE_MONSTER` User Special Access¶

Now with everything centralized I want to make things a little bit more difficult. In `security.yml` I've given the `leanna` user a special role called `ROLE_COOKIE_MONSTER`:

```
# app/config/security.yml
security:
    # ...

    providers:
        in_memory:
            memory:
                users:
                    ryan:  { password: cookie, roles: 'ROLE_COOKIE_ENJOYER' }
                    leanna: { password: cookie, roles: 'ROLE_COOKIE_MONSTER' }
```

If you have this role, I want to make it so you can eat any cookie even if you didn't bake it. Seems like a jerk thing to do but let's try it out.

To do this, we basically want to call the `isGranted` function on the security system from inside of our voter. Now, out-of-the-box we don't have access to do this, so we're going to need to do a little bit of dependency injection. If you're thinking that we'll inject the `security.context`, you're basically right. The only issue is that because we're inside of the security system if we try to inject the security system into here we're going to get a circular dependency. Instead, I'm going to inject the entire container, which, yes is typically a bad practice, but in this case we can't avoid it and it's not going to kill us:

```
// src/AppBundle/Security/CookieVoter.php
// ...

class CookieVoter extends AbstractVoter
{
    private $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }

    // ...
}
```

Head back to `services.yml` add an arguments key now that we have a `__construct` function and use `@service_container` to inject the entire container:

```
# app/config/services.yml
services:
    app_cookie_voter:
        class: AppBundle\Security\CookieVoter
        arguments: ["@service_container"]
        tags:
            - { name: security.voter }
```

Back down in `isGranted` we can easily add the logic we need:

```
// src/AppBundle/Security/CookieVoter.php
// ...

protected function isGranted($attribute, $object, $user = null)
{
    if (!is_object($user)) {
        return false;
    }

    // in 2.5 and earlier, use this:
    // $this->container->get('security.context');
    // security.context exists in 2.6, but is deprecated
    $authChecker = $this->container->get('security.authorization_checker');

    if ($authChecker->isGranted('ROLE_COOKIE_MONSTER')) {
        return true;
    }

    if ($object->getBakerUsername() == $user->getUsername()) {
        return true;
    }

    return false;
}
```

Now I'm using Symfony 2.6 which gives us a brand-new service called `security.authorization_checker`. This is actually a new service for Symfony 2.6. Before it was known as `security.context`. Now don't worry because `security.context` still exists and will still exist until Symfony 3.0. So if you're on Symfony 2.6 use the new service name. If you're on 2.5 or earlier just use `security.context`. The nice thing is that both of them have the same `isGranted` function on it which we can use now to check to see if the user has the `ROLE_COOKIE_MONSTER` role. If they do, let's give them access.

When we try it out there's no difference and that's a good thing. I'm logged in as `ryan` so I don't actually have this role. So I'll logout. Let's login as `leanna`, password `cookie`, and.....COOKIES FOR EVERYBODY!

Adding Multiple Actions (NOM, DONATE) to 1 Voter¶

I want to do one more crazy thing. Let's pretend like we want to be able to donate our cookies to friends. Now I know that's crazy why would you donate cookies to other people? But let's just try it out. I don't actually have the logic for this but that's okay. Let's go into `index.html.twig` and add a link for this. We're just going to see if we can get the link to hide and show correctly:

```
{# app/Resources/views/Cookie/index.html.twig #}
{# ... #}

{% for cookie in cookies %}
    {# ... #}

    <td>
        {% if is_granted('DONATE', cookie) %}
            <a href="">Donate</a>
        {% endif %}
    </td>

    {# ... #}
{% endfor %}
```

Just like before I'm inventing this `DONATE` string. If we don't do anything else and refresh, we'll actually see that the link doesn't show up. If no voters vote on our attribute, then by default it's going to return false. Now why is our voter not voting on it? Because of the `getSupportedAttributes` function.

Let's update that to return true for both `NOM` and `DONUT` ...I mean `DONATE`:

```php
// src/AppBundle/Security/CookieVoter.php
// ...

protected function getSupportedAttributes()
{
    return array('NOM', 'DONATE');
}
```

Now `isGranted` is going to be handling two different attributes, `NOM` and `DONATE`. This is the perfect situation for everyone's beloved switch case statement. So let's set that up, and we have two cases one for `NOM` and one for `DONATE`. And the logic for `NOM` is exactly what we had before so I'll just copy that, paste that up and if it doesn't get into either those if statements we'll return false:

```php
protected function isGranted($attribute, $object, $user = null)
{
    if (!is_object($user)) {
        return false;
    }

    $authChecker = $this->container->get('security.authorization_checker');

    switch ($attribute) {
        case 'NOM':
            if ($authChecker->isGranted('ROLE_COOKIE_MONSTER')) {
                return true;
            }

            if ($object->getBakerUsername() == $user->getUsername()) {
                return true;
            }

            return false;
        case 'DONATE':
            // todo ...
    }

    return false;
}
```

For the `DONATE` case, again, we can do literally anything we want to inside of this. If we want to go out and make crazy database queries to figure out something we can do that. In our case since chocolate cookies are the most delicious, let's only give away cookies that aren't chocolate. So, for my crazy business logic I'm just going to see if the word chocolate appears in the name of the cookie. If it does I'm not going to give it away. But if it doesn't you can have it:

```php
switch ($attribute) {
    case 'NOM':
        // ...
    case 'DONATE':
        return stripos($object->getFlavor(), 'chocolate') === false;
}
```

At the bottom of this function, I still have this false here. This should technically never get hit. Even if we pass something other than `NOM` or `DONATE` to `isGranted` Symfony is not going to call our voter because of the `getSupportedAttributes`.

So, you can put anything down here I like to throw an exception just incase something insane happens. But you're going to be fine either way:

```
protected function isGranted($attribute, $object, $user = null)
{
    // ...

    switch ($attribute) {
        // ...
    }

    throw new \LogicException('How did we get here!?');
}
```

Cool, let's see which cookies we can giveaway. This time we see the donate link only next to the cookies that aren't chocolate. That's perfect.

Let's use some Constants¶

Now, some of you may be thinking that I'm crazy for having these strings like `NOM` and `DONATE` all over my application. And actually, I agree with you. Normally whenever I have a naked string somewhere I make it a constant instead. So in this case I'll create two constants: `ATTRIBUTE_NOM` and `ATTRIBUTE_DONATE`:

```
// src/AppBundle/Security/CookieVoter.php
// ...

class CookieVoter extends AbstractVoter
{
    const ATTRIBUTE_NOM = 'NOM';
    const ATTRIBUTE_DONATE = 'DONATE';

    // ...

    protected function getSupportedAttributes()
    {
        return array(self::ATTRIBUTE_NOM, self::ATTRIBUTE_DONATE);
    }

    // ...

    protected function isGranted($attribute, $object, $user = null)
    {
        // ...

        switch ($attribute) {
            case self::ATTRIBUTE_NOM:
                // ...
            case self::ATTRIBUTE_DONATE:
                // ...
        }

        throw new \LogicException('How did we get here!?');
    }
}
```

Then we can use these inside of `getSupportedAttributes` and later we can use it inside of the `isGranted` function. This helps out with typos but it also allows us, if we want to, to put some PHP documentation above those constants so future us can come and read what nom and donate actually mean.

We can also go into our `CookieController` and use the constant there:

```php
// src/AppBundle/Controller/CookieController.php
// ...

if (!$this->isGranted(CookieVoter::ATTRIBUTE_NOM, $cookie)) {
    throw $this->createAccessDeniedException(
        'You did not bake this delicious cookie!'
    );
}
```

And yes we can also use the constants inside of the twig template with twig's `constant()` function, but honestly it's kind of ugly so for me I just keep the strings here.

Go Security Voters Go!¶

So security voters are all about solving that case when you need figure out if a user has access to do something to a specific object. They help to keep your template logic and your controller logic really simple and they're one of my favorite features. So try them out and let me know what you think.

Symfony also has an ACL system but it's incredibly complex and I only recommend that you use it if you have really complex authorization requirements. If you can somehow write a few lines of code to figure out if a user has access to do something do that in a voter don't worry about ACL.

Alright see you guys next time!