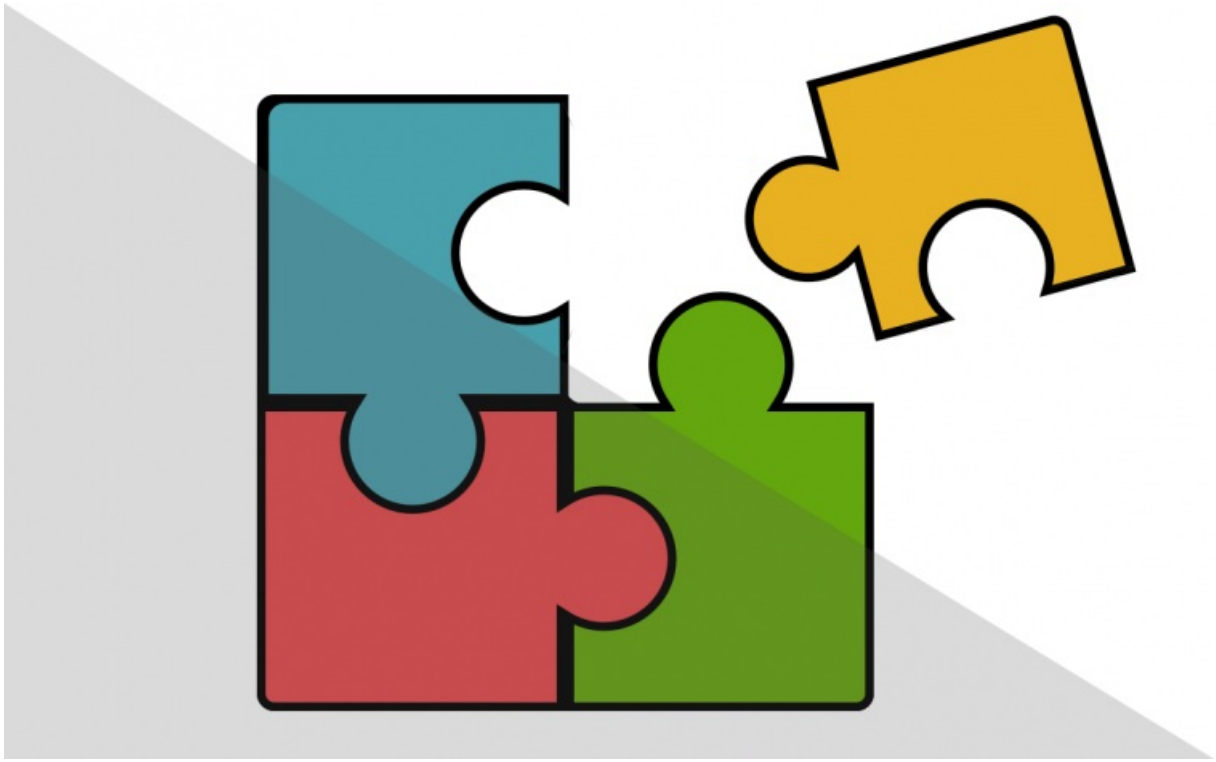
Contributing back to Symfony!



With <3 from SymfonyCasts

Chapter 1: Organization & Triaging

What could be better than eating ice cream at the beach? Only one thing I can think of: eating ice cream at the beach... wait for it... *while* contributing to Symfony!

The Massive Power of Contributors

Seriously, I am *super*, *duper*, double-duper, excited about this tutorial! If you're like me, you probably use Symfony almost every day. And that means, we're taking advantage of *countless* hours of volunteer work from *thousands* of people! Contributing to Symfony is a great way to give *back* and become part of that amazing effort.

But, I also have a few other motivations. Like, if you want to *truly* become an expert on one part of Symfony, there is *no* better way than reviewing a pull request or fixing a bug. Seriously.

Or, have you ever been annoyed by part of Symfony and wanted to improve it? How about this: have you ever been confused, *finally* figured something out, and then realized a *small* change to the documentation could have saved you hours of frustration?

These are the things that get me excited to contribute back to Symfony! How cool is it that *you* could save other people *hundreds* of hours by improving the documentation with extra information that the core team didn't realize was missing? Symfony is *truly* community-driven. There's actually no official roadmap: if you want to add something, do it!

The Organization of Symfony

Excited? There's just one small problem: contributing... ain't easy! At least, not at first. Symfony is a *huge* and complex project. But, you will *not* regret learning how to give back. It's fun and will make you an even better developer.

Let's jump in! The main repository for Symfony lives at https://github.com/symfony/symfony. This holds *almost* all of the Symfony libraries. There are a *few* others that live in other places - but we'll talk about those later.

And, woh! 749 issues and 181 pull requests! That's, ah, a *lot*! And this leads me to the *first*, *most* important and *least* celebrated way of contributing: triaging! Here's the truth: there are too many issues and too many pull requests for the Symfony core team to be able to reply & review everything.

The first way to contribute is to "triage": find an issue and help push it forward. If it's a feature idea, you can give your feedback or offer alternate solutions. If it's a bug, see if you can replicate that bug. We're going to do this.

You can also triage pull requests: find one, review its code, give your opinion on whether or not you think it's a good idea, and even test it in a real project to make sure it works. We'll do that too!

Oh, and I recommend focusing on *newer* issues and pull requests, at least at first. If a PR or issue is old, it's probably pretty complicated.

Your Opinion is Respected

If reviewing code or giving your opinion in a big repository like Symfony sounds scary, don't worry! Symfony is a friendly place: everyone has the same goal: to help move the project forward. Sure, it *is* possible that you'll say something that's not completely correct. I do that all the time! I think I'm kinda famous for it! It's really no big deal. Honestly, the time that you took to review that pull request or issue has a *high* value. And if you say something that isn't totally right, someone else will nicely correct you, you'll learn something, and the whole project will move forward. Be nice, don't be afraid to be wrong, and use any feedback as a way to learn more.

Reviewing a Pull Request

So, let's start contributing! Let's triage a pull request that I found: - it's number 28069. This PR is from my friend Colin, who's proposing a new `MultipleOf` validation constraint that checks whether a submitted value is divisible by another number.

I like this idea, but this PR hasn't gotten *any* attention yet. This is a perfect opportunity for us to help push it forward!

First, let's review the code. As a new contributor, you might not really know what to look for when reviewing. No problem: just see if the code makes sense and look for potential bugs or other issues. You don't have to be perfect: every little bit helps.

To create a validator, you need two classes. The first represents the annotation: `MultipleOf`. The second - `MultipleOfValidator` - is the class that actually does the validation work.

The annotation has an option message:

> The value should be a multiple of {{ compared_value }}

That's a pretty good message. In the validator, Colin uses `` `fmod` `` to compare the values, which means the user can compare decimals - like 1 is a multiple of 5. Yea, this all looks pretty good to me!

The second thing to look for is if the PR has a test: *most* features need some. And, no surprise, Colin did a great job here too: he's testing valid and invalid comparisons. This test uses a special base class to hook this all together.

So... I have *no* comments to add to this pull request! And even *that* is valuable! We'll be able to post that we reviewed the code and it looks good to us. But, there *is* still one important question: does this... *actually* work? It's one thing to look at the code, but it helps *so* much if someone in the community says:

> Hey! I actually *tried* this in a real project and it works great!

Let's *be* that wonderful person next!

Chapter 2: Testing the Code from a PR

It looks like Colin did a *great* job with this new feature. But, we can give a merger *much* more confidence by actually testing it in a real project!

Creating a new Test App

In PhpStorm, I've already created an empty `contributing` directory. And, I already have a terminal open to this same place. To test the PR, let's literally create a brand-new Symfony app:

```
$ composer create-project symfony/skeleton
```

I'm using `symfony/skeleton` instead of the larger `symfony/website-skeleton` to keep things as small and focused as possible. Grab the `dev-master` version of the skeleton:

```
$ composer create-project symfony/skeleton:dev-master
```

Why? Colin's PR is against Symfony's `master` branch. So, to test it, I want to create an app that's based on that *same* version of Symfony.

Finally, put this into a new directory called `triage_pr_28069`:

```
$ composer create-project symfony/skeleton:dev-master triage_pr_28069
```

When that finishes, move over and... yea! Here's the new app. Check out its `composer.json` file:

62 lines | triage_pr_28069/composer.json

```
{
... lines 2 - 4
    "require": {
... lines 6 - 8
        "symfony/console": "^4.2",
... line 10
        "symfony/force-lowest": "=4.2",
        "symfony/framework-bundle": "^4.2",
        "symfony/yaml": "^4.2"
    },
    "require-dev": {
        "symfony/dotenv": "^4.2"
    },
... lines 18 - 60
}
```

It's using version 4.2 of Symfony, which is the next, unreleased, version of Symfony at this moment. In other words, this code is from Symfony's master branch. We also have `minimum-stability` set to `dev`:

62 lines | triage_pr_28069/composer.json

```
{
... lines 2 - 3
    "minimum-stability": "dev",
... lines 5 - 60
}
```

Which means that Composer will try to install new, unreleased version of libraries.

Look back at the PR: all of the changes were to the `Validator` component. Ok, let's get that installed: find your terminal, move into the directory and run:

```
$ composer require validator
```

This will install the `dev-master` version of `symfony/validator`. In other words, it will get the code from Symfony's `master` branch. But... hmm... that's not quite what we want: we somehow need to get the code from *Colin's* branch. How can we do that? Oh, it's *super* cool.

Getting the Code from the Pull Request

Go to your terminal and open a new tab. Go back up to the `contributing` directory. I'm going to clone the *entire* Symfony project into a new directory here. To do that, go back to your browser, move to the repository's homepage, click "Clone or download" and copy the URL.

Move back over, `git clone` and paste:

```
$ git clone git@github.com:symfony/symfony.git
```

When that finishes, we now have a `symfony` directory right next to our app. To get Colin's branch, we have a few options. Move into the new `symfony` directory:

```
$ cd symfony/
```

And type:

```
$ git remote
```

and

```
$ git remote show origin
```

Let's add a *second* remote for Colin's *fork*. Copy the Symfony URL, then run `git remote add` and paste. Copy Colin's username - `colinodell`, move back, call the new remote `colinodell`, and change the username part of the URL:

```
$ git remote add colinodell git@github.com:colinodell/symfony.git
```

Nice! Grab his branches with:

```
$ git fetch colinodell
```

Yep! There's the branch: `feature/multiple-of-validator` - this is the one used for the PR. To check out to that code, create a new branch:

```
```

```
$ git checkout -b feature/multiple-of-validator colinodell/feature/multiple-of-validator
```

Sweet! To prove we've got the right code, go back to PhpStorm, press `Shift` + `Shift`, and search for the new file. There it is!

We *now* have a test app *and* the new code in our `symfony` directory. But, they're not connected yet! Let's do that next.

Chapter 3: Linking Symfony deps to your Local Copy

Here's the question now: how can we make our test app use the pull request code from the `symfony/` directory? Check out the `vendor/symfony` directory in the app: it's just a bunch of sub-directories, each containing code from Symfony's master branch. But, what we *really* want is for this `validator` directory to instead be a symbolic link to the correct path in our `symfony/` directory: `src/Symfony/Component/Validator`.

We could do this by hand... but! Symfony has a cool script to do this automatically: it's called `link`.

Go back to the terminal that holds the app and run:

```
$ ls -la vendor/symfony
```

Yep! Just a bunch of lonely directories. Go back to the other tab and run `./link` and point to our app: `../triage_pr_28069`:

```
$ ./link ../triage_pr_28069/
```

Wow! Go *back* to your app and check the symfony directory again:

```
$ ls -la vendor/symfony
```

Awesome! Every package that comes from the `symfony/symfony` repository is now a symlink to our local copy! In other words, our app is *now* using Colin's code!

Testing the PR Code!

It's time for us to write some code that tests the new validator! In `src/`, create a new PHP class, um, how about: `ClassToValidate`:

15 lines | triage_pr_28069/src/ClassToValidate.php

```
... lines 1 - 2
namespace App;
... lines 4 - 6
class ClassToValidate
{
... lines 9 - 13
}
```

I'm feeling creative!

Inside add a new public property called `$enteredNumber`:

15 lines | triage_pr_28069/src/ClassToValidate.php

```
... lines 1 - 2
namespace App;
... lines 4 - 6
class ClassToValidate
{
... lines 9 - 12
    public $enteredNumber;
}
```

I'm trying to keep my code as *simple* as possible: a public property is a nice shortcut.

Next, add the annotation: `@Assert\MultipleOf()` of 10. I'm also going to add a *second* annotation that will eventually fail - `@Assert\Blank()` - just to make sure everything is working ok:

15 lines | triage_pr_28069/src/ClassToValidate.php

```
... lines 1 - 2
namespace App;

use Symfony\Component\Validator\Constraints as Assert;

class ClassToValidate
{
    /**
     * @Assert\MultipleOf(5)
     * @Assert\Blank()
     */
    public $enteredNumber;
}
```

To try this, in the `Controller/` directory, create a new class: `TestingController`. Fill in the namespace: `App\Controller`:

18 lines | triage_pr_28069/src/Controller/TestingController.php

```
... lines 1 - 2
namespace App\Controller;
... lines 4 - 7
class TestingController
{
... lines 10 - 17
}
```

Because we have multiple apps in one PhpStorm project, some of the magic we normally get isn't working. Inside this, add `public function test()`. Ok: to test validation we'll need the validator service *and* the object to validate. And an argument: `ValidatorInterface $validator`:

18 lines | triage_pr_28069/src/Controller/TestingController.php

```
... lines 1 - 2
namespace App\Controller;
... lines 4 - 5
use Symfony\Component\Validator\Validator\ValidatorInterface;

class TestingController
{
    public function test(ValidatorInterface $validator)
    {
... lines 12 - 16
    }
}
```

Then, `$myObject = new ClassToValidate()` and set its `enteredNumber` to 10:

18 lines | triage_pr_28069/src/Controller/TestingController.php

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

use App\ClassToValidate;
use Symfony\Component\Validator\Validator\ValidatorInterface;

class TestingController
{
    public function test(ValidatorInterface $validator)
    {
        $myObject = new ClassToValidate();
        $myObject->enteredNumber = 10;
... lines 14 - 16
    }
}
```

Oh, and change the `MultipleOf` to be 5:

15 lines | triage_pr_28069/src/ClassToValidate.php

```
... lines 1 - 6
class ClassToValidate
{
    /**
     * @Assert\MultipleOf(5)
... line 11
     */
    public $enteredNumber;
}
```

We'll test that 10 is a multiple of 5.

To validate, add `$errors = $validator->validate($myObject);`. Then, dump that and die!

18 lines | triage_pr_28069/src/Controller/TestingController.php

```
... lines 1 - 7
class TestingController
{
    public function test(ValidatorInterface $validator)
    {
        $myObject = new ClassToValidate();
        $myObject->enteredNumber = 10;

        $errors = $validator->validate($myObject);
        var_dump($errors);die;
    }
}
```

Finally, we need a route for this! We don't have annotations installed, so, to keep things simple, add this in `routes.yaml`: uncomment the example, and change the controller to `TestingController::test`:

4 lines | triage_pr_28069/config/routes.yaml

```
index:
    path: /
    controller: App\Controller\TestingController::test
```

Done! Back in the terminal, start the built-in PHP web server in the `public/` directory:

```
$ php -S localhost:8000 -t public
```

We're ready! Find your browser, go to http://localhost:8000 and... what?! No validation errors! That's actually *not*

good - the `@Blank` constraint should give us *one* error:

15 lines | triage_pr_28069/src/ClassToValidate.php

```
... lines 1 - 6
class ClassToValidate
{
    /**
... line 10
     * @Assert\Blank()
     */
    public $enteredNumber;
}
```

The problem is that our setup is not quite complete: to use annotations with the validator, you need to install the annotations library. Stop the web server and run:

```
$ composer require annotations
```

This installs `sensio/framework-extra-bundle` ... we only *technically* need to install `doctrine/annotations`. But, that's ok. Restart the built-in web server:

```
$ php -S localhost:8000 -t public
```

Move over and, refresh. It works! We get the *one* expected error from `@Assert\Blank`, but we do *not* get a second error: 10 *is* a multiple of 5. To make sure the failure works, change this to 9:

18 lines | triage_pr_28069/src/Controller/TestingController.php

```
... lines 1 - 7
class TestingController
{
    public function test(ValidatorInterface $validator)
    {
... line 12
        $myObject->enteredNumber = 9;
... lines 14 - 16
    }
}
```

move over and... yes! There is the second error. That error language looks really nice to me.

Finishing the PR Review

Hey! The code works! This is great news! Let's go back to GitHub and tell the world! I'll "Approve" this pull request. And, like everything, don't worry: this doesn't mean that the PR is *definitely* perfect: just that you think it's ready. The *really* important part is to add as much information about *why* you think it's ready or not ready. In this case, we checked the code *and* we *actually* tested this in a real project.

And... approve! Oh, but I *did* forget to check one thing: whether or not this new feature has a documentation pull request or issue. But, of course, it does! Not *every* pull request needs documentation - but, if you think it does, and it's missing, gently poke the pull request's author. Or, even better! Go create the documentation pull request yourself! We'll do that later.

Oh, and fun fact! This feature was *merged* about 1 week after we did this review. Go open source! After some good community feedback, it was renamed from `MultipleOf` to `DivisibleBy`. A great change!

Next, let's triage an issue and try to close a bug.

Chapter 4: Triaging a Bug Issue

We just reviewed our first pull request! So let's see what other trouble we can get into! One of the *most* important thing you can do is to triage *issues*.

If an issue is for a feature request, then it's probably a discussion about whether or not it's a good idea and the best implementation. Helping those discussions is great. But... click on the "bug" label. *These* are where you can *really* help.

Symfony is pretty stable & complex. So, if there *is* a bug, it's usually pretty complicated or involves some edge-case situation. These can take a lot of time to understand and replicate. The core team *really* needs help from the community to verify the bug, ask for more information from the user and, ultimately, to create a "reproducer": a tiny app that clearly shows that bug in action.

Triaging an Issue

Let's triage an issue I found a few days ago: #27901. Ok, the user says that he got an error when trying to serialize a Doctrine QueryBuilder with the web profiler: something about not being able to serialize a PDO instance.

The web profiler works by collecting a *bunch* of information about the request and serializing it to a cache file. It looks like something failed during that process. This is actually a pretty nice bug report because he lists the steps to reproduce: install the web profiler and then call `execUpdate` on Doctrine's lower-level `Connection` object, passing it a QueryBuilder. He even suggests a solution!

Ok, so, how can we help? First: see if we can reproduce & understand the issue. Let's create another small project for this. Notice that this is an issue that's reported on the stable version of Symfony: 4.1. So, we should create a new app based on that same version - not `dev-master` like before.

Press `Ctrl` + `C` to stop the server, move back up to the top `contributing` directory and run:

```
$ composer create-project symfony/skeleton triage_issue_27901
```

Because we're not specifying a version, it will use the current stable version: 4.1.

When this finishes, move into the directory. To replicate this bug, we will at *least* need to install the stuff he's using: Doctrine and the web profiler. Back at the terminal, just install Doctrine for now so we can write some code:

```
$ composer require orm
```

And... done! Close the old directory and open this new one. Then, go straight to create a new controller class: how about `Issue27901Controller`. Give it a `public function test()`:

11 lines | triage_issue_27901/src/Controller/Issue27901Controller.php

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

class Issue27901Controller
{
    public function test()
    {

    }
}
```

Ok: check back on the issue. He's using the Doctrine Connection object - a lower-level object I don't use too often.

To see out how to get it, find your terminal and run:

```
$ php bin/console debug:autowiring
```

and scroll up. Yep! It looks like we can type-hint a `Connection` class to get the service we need. Do that: `Connection $connection`:

14 lines | triage_issue_27901/src/Controller/Issue27901Controller.php

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

use Doctrine\DBAL\Connection;

class Issue27901Controller
{
    public function test(Connection $connection)
    {
... lines 11 - 12
    }
}
```

Next, he calls `execUpdate()` and passes it a `QueryBuilder` argument. You may already be familiar with the `QueryBuilder` from Doctrine. Well, in this case, because we're using the lower-level `Connection` class from the Doctrine DBAL library, the `QueryBuilder` is *also* a lower-level class from that library.

These are the types of little details that can make triaging a bug tough! But, it's also part of the fun: you'll need to *really* dig into the code to find out what's going on.

Create the QueryBuilder with `$connection->createQueryBuilder()`:

14 lines | triage_issue_27901/src/Controller/Issue27901Controller.php

```
... lines 1 - 6
class Issue27901Controller
{
    public function test(Connection $connection)
    {
        $qb = $connection->createQueryBuilder();
... line 12
    }
}
```

I won't even do anything with it yet: we're still investigating. Next, he calls `execUpdate()`. Oh, but that doesn't exist! I bet he meant `executeUpdate()` - pass that `$qb`:

14 lines | triage_issue_27901/src/Controller/Issue27901Controller.php

```
... lines 1 - 6
class Issue27901Controller
{
    public function test(Connection $connection)
    {
        $qb = $connection->createQueryBuilder();
        $connection->executeUpdate($qb);
    }
}
```

Great! At this point, I would normally install the web profiler, create some database entities and use a real query in the controller to see if we can replicate the error. But, before we do that, I noticed something: the first argument looks like it's supposed to be a string! Hold `Command` or `Ctrl` and click to open the `executeUpdate()` method.

Yep! The first argument should be a string! But, the user is passing a `QueryBuilder` object! In other words, I don't think this is a bug! The *only* reason the user's query actually works is that, if you open the `QueryBuilder` class, it

has a `__toString()` method. Doctrine is probably accidentally converting this object to a string and using that SQL.

This is why his possible solution is to, inside a related class, convert the `sql` - which is a `QueryBuilder` in his case - into a string. That *would* fix things, but I don't think this is really a bug.

But even still, it's *awesome* that the user opened this issue. In a lot of cases, *even* if there is no bug, we can use the mistake to improve things, like with better error messages.

Replying to the Issue

So, let's reply! And give as *many* specific reasons why we think this might not be a bug: in this case, that the first argument expects a string.

As extra credit, I'll link to this exact code. Go to the `doctrine/dbal` repository. Then, press the letter `t` to open this search screen. I *live* by this shortcut. Look for `Connection.php` and open it. Search for `executeUpdate()` and... click to select that line: this updates the URL to point here.

Then - here's another trick - press the `y` key. This changes the URL from `master` to the actual commit sha. This helps make sure that this link - to line 1068 - will *forever* point to the line we want - even if someone makes changes on the `master` branch and moves this line.

I'll paste the link and add a few more details. I *really* try to be as friendly as possible: this is our chance to help make Symfony a warm & welcoming community. Even if this is *not* a valid issue, it's great the user took their time to help report it.

And... boom! You probably won't have the power to close the issue, but this should make it easy for someone else to do that. Achievement unlocked!

This bug turned out to *not* be a bug. So, let's hunt for a bug that really *is* a bug. And learn how to create and share a "reproducer" project... which is seriously *almost* as valuable as actually fixing the bug.

Chapter 5: Bug Reproducer

Triaging issues and pull requests is seriously, the *best*. But, occasionally, *you* might be the person who finds the bug! That happened to me just today, and I want to report it!

To make the *best* possible bug report, we should create a "reproducer": a Symfony project that shows the bug with as *little* code as possible. I don't have the error I saw in front of me now, but it was pretty simple: I created a form class, tried to use it in my controller, then boom! I got a *very* strange container cache error.

Creating the Reproducer Project

The bug happened when I was playing with the `master` branch of Symfony. So, let's create our new reproducer project based on that:

```
$ composer create-project symfony/skeleton:dev-master container_bug_reproducer
```

When that finishes, move into the new directory:

```
$ cd container_bug_reproducer/
```

The *one* package I know I need is `form`. Get it installed:

```
$ composer require form
```

While we're waiting for this to finish, go back to PhpStorm, close a few files, and go into the new directory. To reproduce my error, I know I need a new form class. Right inside `src/`, create a new PHP class: `SomeFormType`. My creativity today is *off* the charts. Make this extend the usual `AbstractType`:

11 lines | container_bug_reproducer/src/SomeFormType.php

```
... lines 1 - 2
namespace App;

use Symfony\Component\Form\AbstractType;

class SomeFormType extends AbstractType
{

}
```

Normally, we would *also* add the `buildForm()` method. But... I'm not even sure that's needed to trigger the bug. So, in the spirit of making our reproducer as *small* and focused as possible, let's skip it, until we *know* it's needed.

In the `Controller/` directory, create another new PHP class: `ContainerTestController` in the `App\Controller` namespace. Give it the usual `public function test()`:

16 lines | container_bug_reproducer/src/Controller/ContainerTestController.php

```
... lines 1 - 2
namespace App\Controller;
... lines 4 - 7
class ContainerTestController extends AbstractController
{
    public function test()
    {
... lines 12 - 13
    }
}
```

Because we need to create a form, extend `AbstractController` from `FrameworkBundle` so we have that shortcut:

16 lines | container_bug_reproducer/src/Controller/ContainerTestController.php

```
... lines 1 - 2
namespace App\Controller;
... lines 4 - 5
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class ContainerTestController extends AbstractController
{
... lines 10 - 14
}
```

Then, `$form = $this->createForm()` and pass `SomeFormType::class`:

16 lines | container_bug_reproducer/src/Controller/ContainerTestController.php

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

use App\SomeFormType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class ContainerTestController extends AbstractController
{
    public function test()
    {
        $form = $this->createForm(SomeFormType::class);
        // error will happen on previous line
    }
}
```

If I'm right, the error will happen *right* here. To test this, find `routes.yaml`, uncomment the route and point the controller to our code:

4 lines | container_bug_reproducer/config/routes.yaml

```
index:
    path: /
    controller: App\Controller\ContainerTestController::test
```

We're ready! Go back to the terminal and start the built-in web server:

```
$ php -S localhost:8000 -t public
```

Move back, find that browser tab and... refresh! Say hello to the super weird error:

> Compile error: failed opening required file

... in some `var/cache/dev` directory. This is related to the container, and it's very low-level code. It's *possible* I'm doing something wrong, but this *really* looks like a bug.

To be sure, let's stop the web server, manually clear the cache directory with:

```
$ rm -rf var/cache/*
```

restart the server and try it again. Same error. Let's take a screenshot: it's *so* useful to be able to see the full stacktrace of an error.

Opening the Bug Issue

Go back to the Symfony repository and click to open a new issue. Yep, this is a "Bug Report". And, this is cool! We have a nice outline of all the info needed. Affected versions: `master`. This *could* be a bug on a stable version, and that *is* something we could check. But, since it would be such a critical and obvious bug, it's probably just on `master`.

For description, let's describe what we're trying to do. I'll also upload my error screenshot. Oh, and I forgot a title: I'll reference part of the error message for this.

Pushing the Reproducer to GitHub

Under "How to Reproduce", ah, *this* is where our issue will *shine*! Let's push our test project up to GitHub so we can share it.

Move back to the terminal, stop the web server, then initialize a new Git repository with:

```
$ git init
$ git add .
$ git commit
```

and a message. Next, go to GitHub, click "New", type a name and click "Create Repository". Copy the two lines near the bottom for an existing repository. Then, find your terminal and... paste!

That's it! Refresh the page on GitHub: here is our simple reproducer app. Copy its URL. Then, head back to the issue. Let's mention our reproducer app first and how to trigger the error on it.

Then, to make life *even* easier, summarize what we did to get the error. As extra credit, I'll even link *right* to the line that triggers it.

And... that's it! I don't have *any* possible solution: this error is way above my pay grade. And, that's ok. Finish the message and... submit!

We're done! *This* is a bug report worth being proud of.

And, to prove it! I have an update! This issue was fixed less than *3* hours after we posted this. That's amazing.

Next, let's jump into how we can contribute *code* to Symfony via a pull request.

Chapter 6: Symfony's Branching Strategy & Pull Requests

We've already helped push forward a pull request, solved an issue and even reported a bug. Hello! We deserve cake!

And we deserve to move up one more level of difficulty: it's time to contribute *new* code with a pull request. Let's look at an issue I found: #27835.

Understanding the Issue

This comes from the Security component. Let me give you some background: if you try to access a protected page as an anonymous user - like `/admin` - Symfony stores that URL to a special key in the session. Then, after you login, Symfony reads this key and redirects the user back to that URL.

Occasionally, it's useful to *manually* set that session key to control where the user goes after logging in. To help with that, Symfony has `TargetPathTrait`. The problem is that, to *use* its `saveTargetPath()` method, you need something called the "provider key"... which is actually just the "name" of your firewall. You *could* hardcode it, but, that *really* shouldn't be necessary.

In a recent version of Symfony, a feature was added so that you can read the firewall name by getting a `FirewallMap` object, calling `getFirewallConfig()` and *then* calling `getName()`. Phew!

But, here's the problem: that `FirewallMap` service is *not* an autowireable service. That makes it inconvenient to use. And, one of the core contributors gives a reason behind why that service is not autowireable.

You *can* work around this. But, I had an idea: I'm not even sure if it's a good idea, but let's try it. What if we created a new `TargetPathHelper` class that allowed you to set this "target path", *without* needing the provider key. Internally, *it* would use the `FirewallMap` to figure it out automatically.

The end user could use this new class without needing to worry about the firewall name at all.

If you don't completely understand, that's ok. The important thing is the *process* we're going to use to bring this new idea to life!

Symfony's Branching Strategy

Go back to PhpStorm: let's change our project to look *only* at the `symfony/` directory. Then, find the terminal that's in this directory. We're still on the feature branch from `colinodell`. Start by making sure your copy of Symfony is up to date by running:

```
$ git fetch origin
```

And then create a new branch for our feature:

```
$ git checkout -b target-path-helper origin/master
```

This is important: we just created a new branch based off of Symfony's `master` branch. Why? Why not base the branch off of Symfony's 4.1 branch - that's the currently-released version?

Let's talk about Symfony's branching system. It's... *kinda* simple. If you're adding a feature, it should *always* be made to the `master` branch. Then, it will be included in the next Symfony *minor* release: in this case Symfony 4.2. But if you're fixing a bug, you should fix that in the *oldest*, *supported* branch where the bug exists. For example,: if you found a bug that was introduced in Symfony 3.4, create your branch based off of `origin/3.4`.

But, if a bug was first found in version 3.2, you actually would *not* base your branch off of Symfony's 3.2 branch.

Why? Because Symfony 3.2 is no longer supported. To help understand this, go to https://symfony.com/roadmap. At this moment, only three versions of Symfony are supported: 2.8, 3.4 and 4.1.

So, if you found a bug in Symfony 3.2, you would fix it in 3.4, because that's the oldest, *supported* version of Symfony that contains the bug. If you found a bug in Symfony 2.7, you would fix it on the 2.8 branch.

But... if I fix a bug in 2.8... won't that bug still exist in 3.4 and 4.1? Ah, a *very* good question. But... no! The core team routinely merges all *old* branches - like 2.8 - up *into* the newer branches, like 3.4 and 4.1. If you fix a bug in 2.8, it will also be merged & included in all newer versions. Booya!

Anyways, because this will be a new *feature*, our branch is based off of `origin/master`. And *now*, we're ready to code. Let's do that next!

Chapter 7: Coding a new Feature

We've just created a new branch based off of Symfony's `master` branch. And now, we're ready to create the amazing new `TargetPathHelper` class. But... where should it live? It's related to Security... which means it could live in the Security component *or* SecurityBundle.

Components Versus Bundles

As a general rule, most code should live in a component so that it's reusable even outside of the framework. But, sometimes, you'll write code that's *really* integrated with the framework. That code will live in the bundle. My best advice... don't over-think it: it usually becomes pretty obvious if you put something in the wrong spot.

Press `Shift` + `Shift` and search for a file that's closely related to our new feature: `TargetPathTrait`. Ok, this lives in the Security component. I'll double click on the directory to move there. At first, it seems like `TargetPathHelper` should live right here. And that's where *I* would put it at first. I say "at first" because, if you started coding, you'd notice a problem.

What problem? This new class will *ultimately* use the `FirewallMap` class internally to do its work. There are *two* `FirwallMap` classes: one lives in the Security component, and the other lives in SecurityBundle. After digging a little bit, you'd find out that *we* will need to use the one from SecurityBundle.

And here's why that's important: a class in a component *can* depend on classes from other components. But, it can *never* depend on a class from a *bundle*. Because our new `TargetPathHelper` needs a class from SecurityBundle, it *can't* live in the component: it must live in the bundle.

If you get this wrong, no big deal: someone will help you out on your pull request.

Go find SecurityBundle and look inside `Security`. Hey! Here are the `FirewallMap` and `FirewallConfig` classes we'll be using! That's a good sign! Create the new PHP class: `TargetPathHelper`. Add our first public function, how about just `savePath()` with a string `$uri` argument:

12 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php

```
... lines 1 - 2
namespace Symfony\Bundle\SecurityBundle\Security;

class TargetPathHelper
{
    public function savePath(string $uri)
    {

    }
}
```

Symfony 4.0 and above requires PHP 7.1, so you *should* use scalar type-hints and return types. But, Symfony does *not* use the `void` return type.

All About PHPDoc

Because this is a public function, we should add some PHPDoc to describe it. Add a clear, but short description above this:

```
26 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 13
class TargetPathHelper
{
    /**
     * Sets the target path the user should be redirected to after authentication.
     *
     * @param string $uri The URI to set as the target path
     */
    public function savePath(string $uri)
    {

    }
}
```

Actually, Symfony does *not* have a lot of PHPDoc... which might seem weird at first. The reason is that we don't want to maintain too much documentation inside the code - we use a separate repository for documentation.

Oh, and, thanks to the `string` type-hint, the `@param` documentation is totally redundant and should be removed... *unless* there's some valuable *extra* info that you want to say about it. I'll keep it and add some extra notes... even though it doesn't add a *lot* of extra context:

```
26 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 13
class TargetPathHelper
{
    /**
... lines 17 - 18
     * @param string $uri The URI to set as the target path
     */
    public function savePath(string $uri)
    {

    }
}
```

Also *every* PHP file in Symfony should have a copyright header on top. Grab that from another file and paste it here:

```
26 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 2
/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Symfony\Bundle\SecurityBundle\Security;

class TargetPathHelper
{
... lines 16 - 24
}
```

Don't worry too much about these details: it's easy to add them later if you forget.

Injecting the Services we Need

To make life nicer, use `TargetPathTrait` on top of the class:

```
30 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 13
use Symfony\Component\Security\Http\Util\TargetPathTrait;

class TargetPathHelper
{
    use TargetPathTrait;
... lines 19 - 28
}
```

Then all *we* need to do is say `$this->saveTargetPath()`:

```
30 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 15
class TargetPathHelper
{
    use TargetPathTrait;
... lines 19 - 24
    public function savePath(string $uri)
    {
        $this->saveTargetPath();
    }
}
```

But... hmm... this needs 3 arguments: the session, provider key - which is the firewall name - and the URI. We know that we can get the firewall name by using the `FirewallMap` service. Let's add some constructor arguments: `SessionInterface $session` and `FirewallMap` - the one from SecurityBundle - `$firewallMap`:

```
46 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 13
use Symfony\Component\HttpFoundation\Session\SessionInterface;
... lines 15 - 16
class TargetPathHelper
{
... lines 19 - 24
    public function __construct(SessionInterface $session, FirewallMap $firewallMap)
    {
... lines 27 - 28
    }
... lines 30 - 44
}
```

I'll press `Alt` + `Enter` and select initialize fields to create those properties and set them:

```
46 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 16
class TargetPathHelper
{
... lines 19 - 20
    private $session;

    private $firewallMap;

    public function __construct(SessionInterface $session, FirewallMap $firewallMap)
    {
        $this->session = $session;
        $this->firewallMap = $firewallMap;
    }
... lines 30 - 44
}
```

Make sure to remove the PHPDoc above each property: this is redundant thanks to the constructor type-hints.

To calculate the provider key, create a new private function: `getProviderKey()` that will return a string. For now, just put a TODO:

```
46 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 16
class TargetPathHelper
{
... lines 19 - 40
    private function getProviderKey(): string
    {
        // TODO
    }
}
```

Back up in `setTargetPath()`, pass `$this->session`, `$this->getProviderKey()` and the `$uri`:

```
46 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 16
class TargetPathHelper
{
... lines 19 - 35
    public function savePath(string $uri)
    {
        $this->saveTargetPath($this->session, $this->getProviderKey(), $uri);
    }
... lines 40 - 44
}
```

Awesome! Look back at `getProviderKey()`:

```
46 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 16
class TargetPathHelper
{
... lines 19 - 40
    private function getProviderKey(): string
    {
        // TODO
    }
}
```

I didn't add any PHPDoc to this function, but that's *not* because I'm lazy. Or... not entirely because I'm lazy. Really, it's for two reasons. First, this is a private function, and those are typically *not* documented inside Symfony. And second, we already have the return type - no reason to duplicate it!

Lets finish this function. To get the firewall name, we need to use the FirewallMap, call `getFirewallConfig()` and pass it the request. Ok: `$firewallConfig = $this->firewallMap->getFirewallConfig()`. But, hmm... we don't have the Request object! No problem: add a third constructor arg: `RequestStack $requestStack`. I'll hit `Alt` + `Enter` again to create that property and set it. Clean off the PHPDoc, then head back down:

```
56 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 13
use Symfony\Component\HttpFoundation\RequestStack;
... lines 15 - 17
class TargetPathHelper
{
... lines 20 - 25
    private $requestStack;

    public function __construct(SessionInterface $session, FirewallMap $firewallMap, RequestStack $requestStack)
    {
... lines 30 - 31
        $this->requestStack = $requestStack;
    }
... lines 34 - 54
}
```

Normally, when you use RequestStack, you call its `getCurrentRequest()` method to get the request. But, in this case, I'm going to use another method: `$this->requestStack->getMasterRequest()`:

```
56 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 17
class TargetPathHelper
{
... lines 20 - 44
    private function getProviderKey(): string
    {
        $firewallConfig = $this->firewallMap->getFirewallConfig($this->requestStack->getMasterRequest());
... lines 48 - 53
    }
}
```

I'm not 100% sure that this is correct. The whole topic of requests and sub-requests is pretty complex. But, basically, Symfony's security firewall only operates on the outer, "master" request. So, to find the active firewall, that's what we should use. If I'm wrong, hopefully someone will tell me on the pull request.

Next, if you look at the `getFirewallConfig()` method, it's possible that this will return `null`. Code for that: if `null === $firewallConfig`:

```
56 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 17
class TargetPathHelper
{
... lines 20 - 44
    private function getProviderKey(): string
    {
        $firewallConfig = $this->firewallMap->getFirewallConfig($this->requestStack->getMasterRequest());

        if (null === $firewallConfig) {
... line 50
        }
... lines 52 - 53
    }
}
```

This is *another* Symfony coding convention: we use Yoda conditionals!

> Mysterious, Symfony's coding conventions are. Herh herh herh.

But hey! If the force isn't strong with you today, don't worry: Symfony has a magic way of fixing coding convention problems that I'll show you later.

If there is no firewall config for some reason, throw a new `LogicException` with as clear a message as possible:

```
56 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php
... lines 1 - 17
class TargetPathHelper
{
... lines 20 - 44
    private function getProviderKey(): string
    {
        $firewallConfig = $this->firewallMap->getFirewallConfig($this->requestStack->getMasterRequest());

        if (null === $firewallConfig) {
            throw new \LogicException('Could not find firewall config for the current request');
        }
... lines 52 - 53
    }
}
```

Why a LogicException? Well, it seems to make sense - something went wrong... logically. And usually, the exact

exception class won't matter. If it *does* matter, someone will tell you when reviewing your PR.

Finally, at the bottom, `return $firewallConfig->getName()`:

56 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php

```
... lines 1 - 17
class TargetPathHelper
{
... lines 20 - 44
    private function getProviderKey(): string
    {
        $firewallConfig = $this->firewallMap->getFirewallConfig($this->requestStack->getMasterRequest());

        if (null === $firewallConfig) {
            throw new \LogicException('Could not find firewall config for the current request');
        }

        return $firewallConfig->getName();
    }
}
```

That should be it!

While we're here, let's add one more function: `getPath()` that will return a string. Inside, return `$this->getTargetPath()` with `$this->session` and `$this->getProviderKey()`:

64 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php

```
... lines 1 - 17
class TargetPathHelper
{
... lines 20 - 47
    public function getPath(): string
    {
        return $this->getTargetPath($this->session, $this->getProviderKey());
    }
... lines 52 - 62
}
```

This time, I *will* add some PHPDoc. I don't need `@return` - that's redundant - but I will add a description about what this method does:

64 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php

```
... lines 1 - 17
class TargetPathHelper
{
... lines 20 - 44
    /**
     * Returns the URL (if any) the user visited that forced them to login.
     */
    public function getPath(): string
    {
        return $this->getTargetPath($this->session, $this->getProviderKey());
    }
... lines 52 - 62
}
```

Making the Class final

And... we're done! Yea, we still need to write a test & add some config to register this as a service: we'll do that next. But, this class should work!

However... I *am* going to make *one*, ahem, final change: add `final`:

64 lines | symfony/src/Symfony/Bundle/SecurityBundle/Security/TargetPathHelper.php

```
... lines 1 - 17
final class TargetPathHelper
{
... lines 20 - 62
}
```

Making this class final means that nobody is allowed to subclass it. Why are we doing this? Because, in the future, if the class is `final`, it will be easier to make changes to it without breaking backwards compatibility. Basically, if you *allow* a class to be sub-classed, you have to be a bit more careful when making certain changes. Making classes `final` is a good "default" for new Symfony classes.

Of course, if there is a legitimate use-case for some to sub-class this, then you don't *need* to make it final. But, while we can easily *remove* `final` later, we can't *add* `final` in the future, at least not without jumping through a few extra hoops to avoid breaking backwards compatibility.

Ok, let's add some service config & a test!

Chapter 8: Writing & Running Symfony's Tests

When you make a pull request to Symfony, you *almost* always need at least one test. And... yea... we definitely need a test for our new `TargetPathHelper`.

But, before we start writing it... shouldn't we *first* figure out how to *run* Symfony's tests? Great idea! And I'm happy to report that it's *quite* easy.

Getting Symfony's Dependencies

Look in the `symfony/` directory. It has a `composer.json` file that describes all of the libraries that Symfony *itself* needs in order to work *and* in order to run its tests.

Move over to your terminal and run:

```
$ composer update
```

There's one important difference between a reusable library like Symfony and a normal application: Symfony does *not* have a `composer.lock` file! We commit the `composer.json` file to Symfony, but we do *not* commit `composer.lock`. Why not? Well, there's just no point. Individual apps that require Symfony will lock Symfony at some version in *their* app. But, when we're working on Symfony itself, we usually want the *latest* version of all of its dependencies.

So before you run your tests, make sure to run `composer update`. Running `composer install` isn't good enough, because there could already be a `composer.lock` file from an *earlier* time you ran `composer install`. Running update makes sure you have the latest stuff for whatever branch of Symfony you're currently on.

Perfect! *Now* we *do* have a `composer.lock` file.

Running Symfony's Test Suite

Ok, we're ready to run the tests! Do it with:

```
$ ./phpunit
```

Um... that's it! This is a wrapper around PHPUnit: it downloads some dependencies to a different directory, then... starts running the tests! And... yea... there are a lot of tests. I'm going to stop these by pressing `Ctrl` + `C`.

Running only Some Tests

To be honest, I *never* run the full test suite locally. You just don't need to! As you'll see in a few minutes, Symfony has a robust continuous integration setup: when you make a pull request, Symfony's test suite is run automatically.

Thanks to that, locally, I usually just run the tests I'm working on. Let's test everything in SecurityBundle:

```
$ ./phpunit src/Symfony/Bundle/SecurityBundle
```

This time... if you didn't fast forward like me... you'd see that these tests only take a minute or two. There *are* a few "skipped" tests: that's probably not something you need to worry about. Some tests require a special PHP extension or some other service that your local computer might not have. So, those tests are skipped. No big deal.

Adding our Test

Now that the tests are running, it's time to add our own! I'll double-click to get back into SecurityBundle. Because we want to test `TargetPathHelper`, the test should live in `Tests/Security`. Create a new PHP class called `TargetPathHelperTest`. Make this extend the normal `TestCase` from PHPUnit:

14 lines | symfony/src/Symfony/Bundle/SecurityBundle/Tests/Security/TargetPathHelperTest.php

```
... lines 1 - 2
namespace Symfony\Bundle\SecurityBundle\Tests\Security;

use PHPUnit\Framework\TestCase;

class TargetPathHelperTest extends TestCase
{
... lines 9 - 12
}
```

Then add `public function testSavePath()`:

14 lines | symfony/src/Symfony/Bundle/SecurityBundle/Tests/Security/TargetPathHelperTest.php

```
... lines 1 - 2
namespace Symfony\Bundle\SecurityBundle\Tests\Security;

use PHPUnit\Framework\TestCase;

class TargetPathHelperTest extends TestCase
{
    public function testSavePath()
    {

    }
}
```

For the body of the test... yea... I'm going to cheat. This isn't a testing tutorial, so I'll paste in some code I already prepared:

39 lines | symfony/src/Symfony/Bundle/SecurityBundle/Tests/Security/TargetPathHelperTest.php

```
... lines 1 - 4
use PHPUnit\Framework\TestCase;
use Symfony\Bundle\SecurityBundle\Security\FirewallConfig;
use Symfony\Bundle\SecurityBundle\Security\FirewallMap;
use Symfony\Bundle\SecurityBundle\Security\TargetPathHelper;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RequestStack;
use Symfony\Component\HttpFoundation\Session\SessionInterface;

class TargetPathHelperTest extends TestCase
{
    public function testSavePath()
    {
        $session = $this->createMock(SessionInterface::class);
        $firewallMap = $this->createMock(FirewallMap::class);
        $requestStack = $this->createMock(RequestStack::class);
        $request = new Request();

        $requestStack->expects($this->once())
            ->method('getMasterRequest')
            ->willReturn($request);

        $firewallConfig = new FirewallConfig('firewall_name', '');
        $firewallMap->expects($this->once())
            ->method('getFirewallConfig')
            ->with($request)
            ->willReturn($firewallConfig);

        $session->expects($this->once())
            ->method('set')
            ->with('_security.firewall_name.target_path', '/foo');

        $targetPathHelper = new TargetPathHelper($session, $firewallMap, $requestStack);
        $targetPathHelper->savePath('/foo');
    }
}
```

Oh, and I need to auto-complete a few things to get the missing `use` statements, like `FirewallMap` from SecurityBundle, and a few other ones:

39 lines | symfony/src/Symfony/Bundle/SecurityBundle/Tests/Security/TargetPathHelperTest.php

```
... lines 1 - 4
use PHPUnit\Framework\TestCase;
use Symfony\Bundle\SecurityBundle\Security\FirewallConfig;
use Symfony\Bundle\SecurityBundle\Security\FirewallMap;
use Symfony\Bundle\SecurityBundle\Security\TargetPathHelper;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RequestStack;
use Symfony\Component\HttpFoundation\Session\SessionInterface;
... lines 12 - 39
```

Our `TargetPathHelper` class doesn't really do much: it pushes most of the work back to the methods from the trait. So, this test basically creates a bunch of mocks, creates a `FirewallConfig` that returns a firewall name of, um, `firewall_name`, and then we ultimately make sure that this special key is set on the session to the URL we passed to `savePath()`.

If you're interested in understanding this test better, you can totally look into it more. But, the beautiful part is that creating a unit test for Symfony is no different than creating a unit test for an application: there's no framework code here.

Let's go run this *one* test directly:

```
$ ./phpunit src/Symfony/Bundle/SecurityBundle/Tests/Security/TargetPathHelperTest.php
```

The *last* step is to register our new class as a service *and* enable it to be autowired. Let's get to it!

Chapter 9: Services, Autowiring & Pushing To GitHub

We *now* have a fully-functional new class *with* a test! But, we have *not* registered this class as a service yet. Which means... the user would still need to do that manually. That's a bummer!

Adding the Service Config

Inside SecurityBundle, look at `DependencyInjection` and open `SecurityExtension.php`. This class loads several XML files that provide all of the services for this bundle. Inside the `Resources/config/` directory, open `security.xml`. Around line 136... yep! You'll see the services that our *new* service depends on - like `FirewallMap` and `FirewallConfig`:

226 lines | symfony/src/Symfony/Bundle/SecurityBundle/Resources/config/security.xml

```
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd"
... lines 6 - 12
    <services>
... lines 14 - 135
        <service id="security.firewall.map" class="Symfony\Bundle\SecurityBundle\Security\FirewallMap">
            <argument /> <!-- Firewall context locator -->
            <argument /> <!-- Request matchers -->
        </service>
... lines 140 - 147
        <service id="security.firewall.config" class="Symfony\Bundle\SecurityBundle\Security\FirewallConfig" abstract="true">
            <argument />                <!-- name -->
            <argument />                <!-- user_checker -->
            <argument />                <!-- request_matcher -->
            <argument />                <!-- security enabled -->
            <argument />                <!-- stateless -->
            <argument />                <!-- provider -->
            <argument />                <!-- context -->
            <argument />                <!-- entry_point -->
            <argument />                <!-- access_denied_handler -->
            <argument />                <!-- access_denied_url -->
            <argument type="collection" /> <!-- listeners -->
            <argument />                <!-- switch_user -->
        </service>
... lines 162 - 223
    </services>
</container>
```

To register our new `TargetPathHelper` as a service, we could include some XML config in *any* of these XML files: it doesn't *technically* matter. But, which file makes the most sense? Well, 1 minute ago, I wasn't sure. But now that I see all of these related services, I think we've *already* found the right place. If we're wrong, someone will tell us when we create the PR.

Add a new service tag. For the id, how about, `security.target_path_helper`. I'm trying to follow the existing naming conventions in this file.

For the class, it's `Symfony`, well, let's cheat: copy the namespace from the class above, paste, then `TargetPathHelper`:

```
226 lines | symfony/src/Symfony/Bundle/SecurityBundle/Resources/config/security.xml
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd"
... lines 6 - 12
    <services>
... lines 14 - 162
        <service id="security.target_path_helper" class="Symfony\Bundle\SecurityBundle\Security\TargetPathHelper">
... lines 164 - 166
        </service>
... lines 168 - 223
    </services>
</container>
```

Inside, our service will need 3 arguments: the session, firewall map & request stack.

Add `<argument type="service" id="session" />`. Next, `<argument type="service" id="" The id for the firewall map is up here: security.firewall.map`. Finally, `<argument type="service" id="request_stack" />`:

```
226 lines | symfony/src/Symfony/Bundle/SecurityBundle/Resources/config/security.xml
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd"
... lines 6 - 12
    <services>
... lines 14 - 162
        <service id="security.target_path_helper" class="Symfony\Bundle\SecurityBundle\Security\TargetPathHelper">
            <argument type="service" id="session" />
            <argument type="service" id="security.firewall.map" />
            <argument type="service" id="request_stack" />
        </service>
... lines 168 - 223
    </services>
</container>
```

Done! Our new class is now registered as a service!

But... there's still *one* small thing missing with this service. To allow `TargetPathHelper` to be *autowired*, like `FirewallMap` in the issue example, we need to create an *alias* from that class to the service id - just like in the comment below.

Enabling Autowiring

To do this, add `<service id="" />`, go copy the *class* name, and paste it here. Then, `alias=""`, copy the service *id* this time, and paste again:

```
226 lines | symfony/src/Symfony/Bundle/SecurityBundle/Resources/config/security.xml
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services/services-1.0.xsd"
... lines 6 - 12
    <services>
... lines 14 - 167
        <service id="Symfony\Bundle\SecurityBundle\Security\TargetPathHelper" alias="security.target_path_helper" />
... lines 169 - 223
    </services>
</container>
```

That's it! The `TargetPathHelper` will now be an autowireable service.

And... we're done! The *last* thing I'd recommend is to create a real project and test your new feature manually. Sure, our *class* has a test... but there is *not* a test for our service config: if we have a typo on the class name, we wouldn't know!

However, because we already went through the process earlier when testing Colin's PR, I'll skip it. But, saying you tested your code in a real app can definitely help push your PR forward.

Hey! We're done with all the hard work! Let's push our code to GitHub!

Pushing your Fork

Head over to the terminal that's in the `symfony/` directory and run:

```
$ git status
```

No surprises! Add everything and then commit with a nice message that briefly describes what we're doing:

```
$ git add .

$ git commit -m "Adding a new TargetPathHelper class and service"
```

Cool!

```
$ git remote
```

Right now, we have two remotes: `colinodell` & `origin`, which is the main `symfony/symfony`. But, of course, we don't have access to push directly to `origin`. Actually, that's *great* - that sounds like *way* too much responsibility to me.

Instead of pushing directly to Symfony, we need to fork the repository. Click "Fork" and either create a new fork, or, if you already have a fork like me, click into it. Here we are: `weaverryan/symfony`.

Next, click "Clone or download", copy the URL, then move back over to the terminal to add this as a new remote: `git remote add weaverryan` and then paste:

```
$ git remote add weaverryan git@github.com:weaverryan/symfony.git
```

Awesome! *Now* we can push. The branch we created is called `target-path-helper`. So:

```
$ git push weaverryan target-path-helper
```

Back to GitHub! If you're lucky, you'll see a little yellow banner about your new branch. This banner doesn't *always* show up, so if it doesn't, you can refresh, find the `target-path-helper` branch and click "New pull request".

Next, let's fill this in & learn about Symfony's continuous integration system *and* the famous... fabbot!

The *first* thing to notice is the base branch. This is *super* important. Because we created our branch off of master, this base branch must be master. If we created the branch off of 3.4, then, of course, this should be 3.4.

One easy way to be sure you have things setup correctly is to check that you only see the commits down here that you expect. If you mess up the base branch, you'll probably see a bunch of extra commits and changes.

Making Changes to your PR

The pull request description comes with a nice template to get us started. For the branch, because this is a new feature, we *do* want the `master` branch. This is not a bug fix and this *is* a new feature.

Oh, and if this is a new feature, apparently, we should update the CHANGELOG. I totally forgot about that!

Go back to your editor. Then, inside whatever bundle or component you're working on - so SecurityBundle for us - find the `CHANGELOG.md` file. If our new feature is accepted, it will be in the next minor version, which is 4.2 right now. You can already see a few new features listed there. Progress!

Let's add *our* new feature: describe what we introduced & why it's useful.

To add this, we *could* just make a second commit. And that would be *totally* fine. In fact, if you're making significant changes to the pull request, making new commits *is* a good idea: it will help people see how your pull request evolves over time.

But, in this case, because the changes are so simple, run:

```
$ git add -u
```

to add everything. Then:

```
$ git commit --amend
```

That will add these changes to my previous commit to keep things clean. Push with:

```
$ git push weaverryan target-path-helper --force
```

The Pull Request Description

Perfect! Head back to the pull request. We did not introduce any backwards compatibility breaks, we didn't deprecate any features and yes, the tests should pass. A lot of these are just reminders to *think* about things. After we submit the PR, we'll see for sure if the tests pass. For fixed tickets, there isn't *always* a fixed ticket, but there is in this case: `#27835`.

Oh, and whenever you add a new feature, you need to create a documentation pull request. I'll put TODO for now. But we *are* going to do this soon.

Finally, at the bottom, it's our job to put a short README so that other people can understand how our feature works. Showing some code examples is best - but because the code behind this is pretty simple, what I *really* want to do is describe *why* this is needed.

And... we're ready! Create that pull request! Boom! Great work people! Will this pull request be accepted? Who

knows? But, we did good work, wrote tests and clearly stated *why* we created this feature. We rock!

Making fabbot Happy

Until... yep! We immediately see failures at the bottom! Two things happen when you create a pull request. First, Symfony's continuous integration system starts running the tests: Travis CI runs tests on Linux & AppVeyor runs them on Windows. Click the details for Travis - it's pretty awesome.

It executes the tests on *multiple* versions of PHP and uses different flags to use both the *newest* version of Symfony's dependencies and the oldest allowed versions. We'll let this keep doing its thing.

The *second* thing that happens after creating a pull request is that you are visited by the famous... fabbot! Fabbot automatically checks your pull request for coding standards violations. Apparently we have two problems! But, here's the best part: copy that curl statement, find your terminal, and paste! Run:

```
$ git diff
```

Cool! This automatically fixed the two whitespace issues that violated Symfony's coding standards. This is why I don't worry *too* much about coding standards until now: fabbot will help us.

And because these changes aren't important, just like before, let's *amend* our commit:

```
$ git add -u
$ git commit --amend
```

Push that with:

```
$ git push weaverryan target-path-helper --force
```

Ok, go back to the pull request. Refresh and... yea! Fabbot is happy!

Caring for your PR After Submitting

So... what now? First, wait to see if the tests pass. If they don't, yea, you'll need to see if our changes caused some unexpected bug. And second, wait for community feedback! Sometimes, feedback can be direct and to the point. We're developers, so we like to look at all the smallest technical details. Don't take this personally: feedback is meant to be constructive - we're all on the same side. And, yea, you'll almost *definitely* need to make at least *some* changes. Heck, I *rarely* make a pull request that doesn't need *significant* changes after some feedback. It's awesome! Usually, someone thinks of a *way* better implementation. When that happens, you know the drill: make the changes, commit, push, and check fabbot and the tests again.

Oh, and don't worry too much about doing the `git commit --amend` thing or rebasing. It's *totally* ok to have multiple commits. And also, when someone merges your pull request, they use a tool that makes it *really* easy to squash all of your commits down into 1 commit, if they want to. That's not something you need to worry about.

Next: we haven't created our documentation pull request yet. Time to do that!

Chapter 11: Uh oh: Documentation Bug!

The *last* TODO for our PR is to create a documentation PR. And, honestly, in my opinion, making changes to the documentation is probably the easiest *and* most *effective* way to contribute to Symfony! There are *tons* of great ways to help the docs, even if you're *not* documenting a new feature.

For example, imagine you're reading the docs - like the forms page. Then, you find something that's inaccurate or confusing. Well, just go back to the top, click "Edit this page", and you'll be inside an editor on GitHub where you can make improvements and create a pull request.

I've worked on the documentation for years. And the *best* way to improve it is to get feedback from real people who are trying to use it. Taking a few minutes to reword a paragraph could save someone else *hours*. That's pretty cool.

Cloning the Docs

Go back to the homepage of the docs. Copy the clone URL: let's clone this down onto our machine. At your terminal, move back into the main contributing directory and run `git clone` and paste.

```
$ git clone git@github.com:symfony/symfony-docs.git
```

I'll also go to PhpStorm and move us back into this main `contributing/` directory so we can see all of the test projects, symfony itself and the new `symfony-docs/` folder.

Hunting down a Bug

Ok: we want to document our new `TargetPathHelper`. Great! Except... where should these new docs live? This can be a real challenge: the docs are huge! If you're not sure, don't worry: just choose some place that makes sense to you. If there's a better place, someone will tell you when reviewing your PR and you can move it!

Head back to your terminal and move into `symfony-docs`. Because this feature builds off of `TargetPathTrait`, let's see where *that's* documented:

```
$ git grep TargetPathTrait
```

Ok: apparently that's covered in some `form_login.rst` file. Go find that in PhpStorm: `security/form_login.rst`. Look *all* the way down at the bottom. Yep, here is where it talks about `TargetPathTrait`:

414 lines | symfony-docs/security/form_login.rst

```
... lines 1 - 396
Redirecting to the Last Accessed Page with ``TargetPathTrait``
------------------------------------------------------------

The last request URI is stored in a session variable named
``_security.<your providerKey>.target_path`` (e.g. ``_security.main.target_path``
if the name of your firewall is ``main``). Most of the times you don't have to
deal with this low level session variable. However, if you ever need to get or
remove this variable, it's better to use the
:class:`Symfony\\Component\\Security\\Http\\Util\\TargetPathTrait` utility::

    // ...
    use Symfony\Component\Security\Http\Util\TargetPathTrait;

    $targetPath = $this->getTargetPath($request->getSession(), $providerKey);

    // equivalent to:
    // $targetPath = $request->getSession()->get('_security.'.$providerKey.'.target_path');
```

We'll add a few more details below this about *our* new class.

But wait! When I first opened this document, I noticed something interesting on top. It describes how this "target path" feature works in general. Then, there's a note below: sometimes redirecting to the originally requested page can cause problems, like if a background AJAX request *appears* to be the last visited page, causing the user to be redirected there:

414 lines | symfony-docs/security/form_login.rst

```
... lines 1 - 26
.. note::

    Sometimes, redirecting to the originally requested page can cause problems,
    like if a background Ajax request "appears" to be the last visited URL,
    causing the user to be redirected there. For information on controlling this
    behavior, see :doc:`/security`.
... lines 33 - 414
```

That makes sense... except, it's not true! Nope, this note is out of date: Symfony *no* longer has this problem. I think I just found a documentation bug!

Let's make sure: go to github.com/symfony/symfony. Then press "t" to open the "file search" and look for a class called `ExceptionListener` from the `Security/` component. *This* is the class that's responsible for setting the `targetPathTrait`. It happens all the way down at the bottom in `setTargetPath()`. If you go to a page like `/admin` as an anonymous user, *right* before you're redirected to the login page, this `setTargetPath()` method is called.

And, cool! This uses the method from `TargetPathTrait`, just like we did. But, check it out, it *checks* to see if the request is an AJAX request - that's the `isXmlHttpRequest()` part. If it *is* an AJAX request, it does *not* set the URL into the session. Yea! The documentation is wrong!

Finding the Correct Bug Branch

The question *now* is: how *old* is this bug? How long ago was this changed in Symfony and what versions of the *docs* do we need to update? Head back to the Symfony Roadmap. The three maintained branches are 2.8, 3.4 and 4.1. Remember: when fixing a bug, you should fix it in the *oldest* maintained branch where the bug exists. The same is true for the docs.

To figure out when the fix was made to Symfony, let's `git blame` this file. Scroll back down to the bottom. Hmm, so this line was last modified two years ago. And if you look at that commit, its changes do *not* include the AJAX part of this line. Yep, the change we're looking for is *more* than two years old. *And* this commit was first included in Symfony 2.7!

In other words, the AJAX fix has existed since Symfony 2.7 or earlier. But, because Symfony 2.7 is no longer maintained, we'll fix this on the 2.8 branch of the docs. Then, after our pull request is merged, our changes will be merged *up* into all of the newer branches by the docs team.

Fixing a Docs Bug

Awesome! Find your terminal and create the new branch:

```
$ git checkout -b remove-outdated-note origin/2.8
```

Move back to the file. Yep, that bad note *did* exist even back then. And, woh! It links to a whole *other* document that describes how to work around this problem. We can delete all of this!

Remove the note first. Then delete that other file:

```
$ git rm security/target_path.rst
```

And... we're ready!

```
$ git status
$ git add -u
$ git commit
```

Describe why we're deleting all this stuff.

Creating the Pull Request

Ok, the code is ready! Head back to GitHub and fork the repository if you haven't already. Then, copy your git URL and add your remote: `git remote add weaverryan` and paste. Now, push!

```
$ git push weaverryan remove-outdated-note
```

Move back and... if you're lucky, you'll see a yellow bar. We *are* lucky this time! Click "Compare and pull request".

Oh, but hmm: why are there two extra commits by other people? Ah, because we need to change our base branch to 2.8.

Much better. In the description, we want to make this as easy as possible to merge. So, let's describe *why* we're removing this and that we checked the code to be sure.

Ok... submit! The docs *also* have a continuous integration system. I want to talk about that next, write our new documentation and learn a bit about the docs format.

After waiting about a minute, oh! You'll notice that the continuous integration for our documentation pull request failed? What does that even mean? Are there tests for the docs?

Let's learn a few important things about the docs. First, you probably noticed that all the files use a `.rst` extension. That's called Restructured text. It's a lot like markdown... on steroids. It has, for example, a special syntax for linking from one page to another - that's this `:doc:` stuff:

```
414 lines | symfony-docs/security/form_login.rst
... lines 1 - 6
Using a :doc:`form login </security/form_login_setup>` for authentication is a
... lines 8 - 10
:doc:`form login configuration reference </reference/configuration/security>` to
... lines 12 - 414
```

Behind the scenes, a build process turns all of this into HTML. But, if we have a link to a document that doesn't exist, that build will fail!

Click "Details" to open Travis CI. The continuous integration system does exactly that: it runs the build to make sure all the basic stuff is okay: syntax, links and a few other things.

And... yep! We have an error: apparently `security.rst` line 1269 contains a reference to a non-existent document `security/target_path`. That's the page we removed! Instead of printing a broken link, we know we need to remove it!

Move back over, find the `security.rst` file and scroll down to line 1269. Ah. This `toctree` thing is another feature of RST - it helps build the table of contents. Remove the `security/target_path` line.

To make sure there aren't any *other* references, find your terminal and search:

```
$ git grep security/target_path
```

Only one other spot - `redirection_map`. That's an internal tool to help us manage old URLs: not something we need to worry about. Let's commit:

```
$ git add -u
$ git commit --amend
```

I'm using amend because this isn't an important change worth making a second commit. Push with:

```
$ git push weaverryan remove-outdated-note --force
```

Hopefully the build will work this time.

Branching for the new Feature

Ok: back to our original task: we need to write documentation for our new feature. That means we need to create a documentation PR against the `master` branch. Go back to to the terminal and create a new branch:

```
$ git checkout -b target-path-helper origin/master
```

Writing in RST

Awesome! Move back and open the `form_login.rst` file again. Scroll all the way down to the bottom.

If you're not comfortable writing documentation *or* if you're a non-native English speaker, you might think that writing docs isn't for you. That's totally not true! The *really* important thing about writing docs is creating good code examples. Pay less attention to writing words and more attention to writing *code* that shows how your feature is used. When you submit your PR, the docs team can help reword & improve the little details. The hard work is writing the code.

I'll start with a quick sentence, then right into the code block:

426 lines | symfony-docs/security/form_login.rst

```
... lines 1 - 392
The last request URI is stored in a session variable named
``_security.<your providerKey>.target_path`` (e.g. ``_security.main.target_path``
if the name of your firewall is ``main``). Most of the times you don't have to
deal with this low level session variable. However, if you ever need to get or
remove this variable, it's better to use the
:class:`Symfony\\Component\\Security\\Http\\Util\\TargetPathTrait` utility::

    // ...
    use Symfony\Component\Security\Http\Util\TargetPathTrait;

    $targetPath = $this->getTargetPath($request->getSession(), $providerKey);

    // equivalent to:
    // $targetPath = $request->getSession()->get('_security.'.$providerKey.'.target_path');

You can also use the ``TargetPathHelper`` service in the same way::
... lines 409 - 426
```

I'll paste in an example I already created:

426 lines | symfony-docs/security/form_login.rst

```
... lines 1 - 407
You can also use the ``TargetPathHelper`` service in the same way::

    // ... for example: from inside a controller
... line 411
    // ...

    public function register(Request $request, TargetPathHelper $targetPathHelper)
    {
        // the user clicked to register: save the previous URL
        if ($request->isMethod('GET') && !$targetPathHelper->getPath()) {
            // redirect to the Referer, or the homepage if none
            $target = $request->headers->get('Referer', $this->generateUrl('homepage'));
            $targetPathHelper->savePath($target);
        }

        // later, after a successful registration POST submit
        return $this->redirect($targetPathHelper->getPath());
    }
```

And, yeah, this green background is super annoying: I don't normally use PhpStorm for documentation. Anyways, a few important things about the format. First, any technical term - like a class name - should be surrounded by two ticks. Second, when you want to add a PHP code block, finish the previous sentence with two colons and indent the code. And third, when you're inside the code, put as *much* context as possible. For example, I've added a note to say that this code is from a controller:

```
426 lines | symfony-docs/security/form_login.rst
... lines 1 - 407
You can also use the ``TargetPathHelper`` service in the same way::

    // ... for example: from inside a controller
... lines 411 - 426
```

You should also be sure to include any `use` statements needed for the new code. Well, we don't include `use` statements for *everything*. For example, I didn't include the `use` statement for the `Request` because people probably know what that is and it's not directly relevant to what we're doing. But, I *did* add the `use` statement for the class we're talking about: `TargetPathHelper`:

```
426 lines | symfony-docs/security/form_login.rst
... lines 1 - 407
You can also use the ``TargetPathHelper`` service in the same way::

    // ... for example: from inside a controller
    use Symfony\Bundle\SecurityBundle\Security\TargetPathHelper;
    // ...
... lines 413 - 426
```

Finally, we recommend *avoiding* big paragraphs of explanatory text. That's why we just included once sentence then code. If you want to explain a bit more, try adding comments *into* the code instead. We've found that people tend to read the code, but skip the paragraphs completely. Use that to your advantage!

And... that's it! Sure, there are a lot of other little format details. But, the docs have *plenty* of examples of how to do just about anything.

Oh, but because this is a new feature, I'll add one more thing. Right above the new text, add a special `versionadded:: 4.2` tag:

```
429 lines | symfony-docs/security/form_login.rst
... lines 1 - 407
.. versionadded:: 4.2
... lines 409 - 410
You can also use the ``TargetPathHelper`` service in the same way::
... lines 412 - 429
```

If our feature is merged, it will be included in Symfony 4.2 - the next Symfony version. This will add a new note highlighting this fact:

```
429 lines | symfony-docs/security/form_login.rst
... lines 1 - 407
.. versionadded:: 4.2
    The ``TargetPathHelper`` class was introduced in Symfony 4.2.

... lines 411 - 429
```

This syntax is also special to RST. You can make tips and notes the same way.

Ok - let's move over, add this file, and commit:

```
$ git add -u

$ git commit -m "Documenting the new TargetPathHelper"
```

And... push:

```
$ git push weaverryan target-path-helper
```

Move back over to GitHub. Hey! The tests passed on our other pull request! Sweet! And, just like always, if you don't see the yellow bar here, go back to your fork, select the new branch and hit "New pull request".

This time, our pull request *should* be against the master branch. I'll prefix the title with `[WCM]` - that means "Waiting for Code Merge" - a little flag to help us know this is for a still-unmerged new feature.

For the body, saying see `symfony/symfony#28181` should be enough. Create that pull request!

Hey! You're now a docs expert! So, I hope to see a *bunch* of docs PR's from you. Do it!

Chapter 13: Recipes & Other Repositories

We've seen a *bunch* of ways to contribute: triaging issues & pull requests, creating pull requests and contributing to the documentation.

But there's so much more! The Symfony ecosystem is a *lot* bigger than just these two repositories. For example, go to github.com/symfony. Woh! There are *118* repositories under Symfony! Click on, for example, `dom-crawler`.

This repository is what's called a "subtree split". Cool name, right? The DomCrawler component *actually* lives and is managed inside of the main `symfony/symfony` repository that we've been working in. I'll show you: open that repository and navigate to `src/Symfony`, `DomCrawler`.

This is the DomCrawler component. An automated process *splits* this directory into its own repository so that people can use it independently. If you want to contribute to DomCrawler, you'll do it in `symfony/symfony`. The sub-tree split is read-only.

How can you know if a repository is a sub-tree split? You'll notice that the "Issues" tab has been disabled.

Other Repositories

Many of those 118 repositories under Symfony *are* subtree splits. And so, you contribute to them in the main repo. But a lot of them are *normal* repositories that you *can* contribute to directly. For example, recipes.

This is one of the *coolest* ways that you can help Symfony. If you install a bundle or library and it doesn't have a good recipe, or doesn't have a recipe at all, you should totally add or improve it! You can create a recipe here, or on the less-stringent `recipes-contrib`.

We're going to improve a recipe in a minute. But, first, I want to point out a few other repositories. Search for "flex". This is the Composer plugin that powers the recipe system. Search for "maker". MakerBundle is all about code generation. Try "encore": this is a library written in Node that helps make Webpack easy. And, one more: panther a new library that allows you to functionally test your pages, including the JavaScript on those pages.

And there are a *lot* more - like `MonologBundle`. The point is: *each* of these needs help triaging issues and reviewing pull requests. And actually, because these independent libraries get less traffic, you can make an even bigger difference in a short amount of time.

Improving a Recipe

Let's make one small contribution to the recipes. Go back to the main recipes repository. One of the recipes is for the `twig/extensions` library: a standalone PHP library. When you install that package, its recipe gives you a new `config/packages/twig_extension.yaml` file. *These* are the four classes provided by that library. After installing the library, you just need to uncomment the ones that you want.

Let's make this even *more* obvious by adding a comment above to describe that.

To do that, go back to main recipes page and copy the clone URL. Hopefully, this process is starting to feel boring... and repetitive. At your terminal, move back into the `contributing/` directory and clone that:

```
$ git clone git@github.com:symfony/recipes.git
```

Then, move inside. To create a pull request, we will eventually need our own fork. I already have a fork, so I'll skip straight to copying *my* URL, going back to the terminal, and adding that remote: `git remote add weaverryan` and paste.

```
$ git remote add weaverryan git@github.com:weaverryan/recipes.git
```

Back in the editor, I'll close a few files. Then open `twig/extensions/1.0/config/packages/twig_extensions.yaml`:

```
11 lines | recipes/twig/extensions/1.0/config/packages/twig_extensions.yaml
services:
    _defaults:
        public: false
        autowire: true
        autoconfigure: true

    #Twig\Extensions\ArrayExtension: ~
    #Twig\Extensions\DateExtension: ~
    #Twig\Extensions\IntlExtension: ~
    #Twig\Extensions\TextExtension: ~
```

Add the comment:

> Uncomment any lines below to activate that Twig extension

```
12 lines | recipes/twig/extensions/1.0/config/packages/twig_extensions.yaml
services:
... lines 2 - 6
    # Uncomment any lines below to activate that Twig extension
    #Twig\Extensions\ArrayExtension: ~
    #Twig\Extensions\DateExtension: ~
    #Twig\Extensions\IntlExtension: ~
    #Twig\Extensions\TextExtension: ~
```

Brilliant! Let's commit this! The recipes repository is a bit unique: it *only* has a `master` branch. So, we'll create our new branch from it:

```
$ git checkout -b adding-twig-extensions-note
```

Then, add, commit

```
$ git add -u

$ git commit -m "Adding a small note about what to do in the twig_extensions.yaml file"
```

and...

```
$ git push weaverryan adding-twig-extensions-note
```

Awesome! Go back to GitHub! Sweet! Here's the yellow bar: click "Compare & pull request". Add a small note about *why* we think this is a good idea. And, make sure we don't have any other "surprise" changes. Looks good. Hit "Create pull request".

Testing a Recipe

This was a pretty simple change. But, when your changes are bigger, you'll probably want to be able to *test* your recipe before it's merged: to see how it works in the real world!

And... we can do that! Almost immediately after posting the PR, you'll hopefully see a message: "Pull request passes validation". This means that our changes passed a few rules that are described in this repository's README.

Another spot says "View deployment". Open that in a new tab. This is *really* cool. The Flex server just "deployed" our recipe. And we can temporarily change our Flex "endpoints" to *use* our new recipe... even though it's not merged yet!

Copy the `export` line, find a terminal and paste:

```
$ export SYMFONY_ENDPOINT=https://symfony.sh/r/github.com/symfony/recipes/449
```

We just set an environment variable on this terminal tab only. To test the recipe, let's just move into one of our projects, like `triage_issue_27901`.

Then run:

```
$ composer require twig/extensions
```

When we run that, it gives us a warning that we're not using the normal Symfony endpoint... which is perfect. And... it looks like it worked!

Go check it out: open `config/packages/twig_extensions.yaml`. We got it! No surprise for *this* small tweak. But for bigger changes, this is *so* useful.

When you're done playing with things, be sure to unset that variable so that you once again use the *real* Symfony endpoint:

```
$ unset SYMFONY_ENDPOINT
```

See you on GitHub!

Ok people, that's it! Oh, there is *so* much good stuff to work on in the Symfony world. And we need the help! There's complex stuff, like working on the main `symfony/symfony` repository. But there are also many, many other ways to contribute, like improving the documentation, working on recipes or just finding that third party library or bundle you love and helping to improve it or its docs. *You* are the person that can make a difference by adding that feature or fixing that bug.

And if you have more questions on contributing, ask them down in the comments! We would love to help answer them

Ok people, seeya on GitHub!