Blackfire.io: Revealing Performance Secrets with Profiling

With <3 from SymfonyCasts

Chapter 1: Performance, Profilers and APMs

Hey friends! Welcome to the *fastest*, most *performant* SymfonyCasts tutorial of all time, on Blackfire. The end. What? We should say a bit more?

Uh, Blackfire is *all* about having fun while you discover ways to make your site *absurdly* fast. We're going to see big graphs, numbers, statistics, animated gifs, and watch all those numbers decrease as we hunt down and eliminate performance bottlenecks. This stuff is just *fun*. And who doesn't want a faster site?

But... ok... *just* "being fun" probably isn't a good enough reason to use Blackfire. If you're trying to "sell" using a tool to your team... or management, the *real* reason is profit. Performance is money. Heck, Google even has a page that will measure the speed of your site and tell you how much *revenue* you can gain by de creasing the rendering time of your site by various amounts.

On the flip side, I'm sure you've heard the famous saying:

> Premature optimization is the root of all evil

I thought it was Nickelback. If that's true... doesn't a having a cool profiling tool like Blackfire make you think *more* about prematurely optimizing? Actually, it's the *opposite*: it let's us focus on creating features and *then* noticing performance problems *if* there are any.

Performance: Server + Network + Rendering

By the way, your site's performance is really three things put together. First, the time it takes your server to build the page. Second, the time it takes to transmit that data over the network. And third, the time it takes for the browser to display stuff - the frontend. You should focus on *all* of these, but the *main* parts are the server and frontend. Your browser has tools to understand and optimize your frontend. Blackfire helps optimize your backend.

Application Performance Monitoring (APM) Versus Blackfire

But it's not the *only* way to monitor performance on your server. The most well-known way is by using an "application performance monitoring tool" - or APM... which is an acronym I had to look up about 10 times before I could remember what it meant! An APM is something that runs on your servers *all* the time, collecting information about load times, slow queries, slow functions, errors and more. The most famous one is probably NewRelic, though Blackfire is planning to release their own sometime soon.

The *great* thing about an APM is that you can see data from *every* request on your production servers. The *bad* part is that, because an APM is always running, it needs to collect data *without* slowing down the page. If it tries to collect too much, *it* would become the performance bottleneck!

Blackfire is a *profiler*. The *big* difference is that, instead of running on *every* single request that our users make... and needing to stay very lightweight, Blackfire only profiles a page when you *tell* it to. It then makes its *own* request to the page and collects an *incredible* amount of *extremely* detailed information. This process *totally* slows down that page load... which is fine, because there's not a real user waiting for it to return.

The *point* is: use an APM *and* a profiler. The APM will give you a constant stream of information from production. The profiler will give you the *deep* information you need when debugging performance on specific pages.

Project Setup

Ok, enough chat! Let's do this! To remove any bottlenecks and maximize your learning performance, you should *totally* code along with me. Download the course code from this page. When you unzip it, you'll find a start/ directory with the same code that you see here. Follow the README.md file for all the setup details. This is a Symfony project - but that won't matter much: we'll mostly focus on understanding and getting the most out of Blackfire.

The last setup step in the README will be to open a terminal, move into the project, use the Symfony binary to start a local web server by typing:

```
symfony serve
```

Ok, let's see the site! Find your browser and head to https://localhost:8000. Now you understand how important this project is. The world has been looking for Big Foot, or "Sasquatch", for years. Thanks to the Big-Foot fanatic community on our site - "Sasquatch Sightings" - we're closer than ever. In our case, better performance doesn't mean more profit, it means, more big foot.

Do... I know where the performance problems are? Nope. No idea. Honestly, I was too focused on getting this site to production to obsess over performance. And... I feel great about that! We'll use Blackfire to find the bottlenecks - if any - and Sasquash them!

Next, let's get Blackfire installed on my local machine and start profiling this local website. And yes, you *can* use Blackfire on production - which is *awesome* - and something we'll do later in the tutorial.

Chapter 2: Blackfire Install: Agent, Probe, Chrome Extension

So let's get Blackfire installed on our local computer. Head over to https://blackfire.io and log in or register for a new account. As you can see, I've been busy using Blackfire already.

Agent & Probe: How it all Works

Click the Docs link on top... then installation on the left. Before we jump in and install everything, I want you to understand just a *little* bit about how this all works: understanding this helped me a *bunch*. If you want to skip this and head to the next video you *can*... just prepare to miss out on some cool diagrams!

Click the "main components of Blackfire" link and scroll down to find... woh! A diagram that shows you *exactly* how Blackfire works.

The Probe: PHP Extension that Collects Data

How about... we look at a simplified version. There are 3 things we need to install. The first is called the "probe", which is really just a PHP extension. You'll install this wherever your code is running - like on your local machine, and later on production. The probe's job is simple, but huge! It's responsible for collecting *all* of the information: all the function calls, how long each took, which function called which other function, how much memory did something take, network requests... you get the idea. By the way, the process of "collecting all the data" is sometimes called instrumentation... which I *only* mention so that if you see this fancy word... it hopefully won't confuse you... it confused me.

The Probe: Collector and Sender

The *second* thing we will need to install is called the "agent". This is a service - or "daemon" - that runs on your computer - or on your production machine. It... just sits there and waits. When the PHP extension - the probe - finishes collecting all the data, it sends that data to the agent. The agent does some processing on it - like removing unimportant information and anonymizing things - then ultimately sends that data to the Blackfire server. It's... the middleman.

So basically, the probe and agent work together to collect the info and send it to Blackfire.

The Browser Extension: Profiling Activator

The *last* piece you'll need to install is a browser extension. Remember: the probe is *not* profiling *every* single request. Normally, when a request comes in, it yawns... and does nothing. The browser extension's job is to *activate* profiling. It basically says:

> Hey probe! Wake up! I'm going to make a request and I *actually* want you to do your thing - collect all the data and sent it to the agent. Cool? Text me when it's done.

And... that's it! This bottleneck-fighting superhero trio is our ticket to performance glory. Next, let's get them installed.

Chapter 3: Installing the Agent, Probe & Chrome Extension

So... let's get these pieces installed! Back on the install page, the setup details will vary based on your operating system. Fortunately, Blackfire has details for pretty much all situations. I'm on a Mac and will use Homebrew to get everything working.

I'll copy the brew tap command, move to my terminal, open a new tab and paste:

brew tap blackfireio/homebrew-blackfire

Installing the Agent

That gives me access to the Blackfire packages. *Now*, install the *agent* - that's the "daemon" that runs in the background - with:

brew install blackfire-agent

Perfect! It says I need to "register" my agent. And... the browser instructions confirm that! I'll copy that command, clear the screen and paste:

sudo blackfire-agent --register

This is going to ask us for our "Server Id" and "Server Token". These are... basically an internal "username and password" that the agent will use to tell the Blackfire servers which *account* the profiles should be attached to. Copy the Server Id, paste, copy the Server Token, paste and... we're good!

Finally, remember how the "agent" is a service that runs in the background? We just *installed* the agent, but it's not running yet. Back in the docs, the next two commands set up the agent as a "service" in Brew, so that it will *always* be running. Copy the first, paste.

ln -sfv /usr/local/opt/blackfire-agent/*.plist ~/Library/LaunchAgents/

Then spin back over again, copy the launchctl load command... and paste that.

launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.blackfire-agent.plist

Cool! If everything worked, the Blackfire agent is now running in the background. You wont really ever see it or care that it's there... but it is... waiting for data.

Installing the Probe

Back on the install docs, the next piece we need is the PHP extension - the probe. Skip this CLI tool for now - we won't need it until later.

To install the PHP extension, we'll once again use brew. But... hopefully you're *not* still using PHP 5.6. Let me head over to my terminal and see what version I'm running:

php --version

7.3.6. Brilliant! So I'll run:

brew install blackfire-php73

Notice that the extension doesn't need *any* authentication info - like a server Id or token. It's beautifully dumb: its job is to profile data, send it to the agent, and let *it* worry about authentication with the Blackfire servers.

We *do*, however, as it says, need to restart our web server. For us, that means going to the other terminal tab, hitting Control + C, and then running

symfony serve

Is the Blackfire extension working? I don't know! Because we're using Symfony, an easy way to check is to hover over the web debug toolbar and click the "View phpinfo()" link. Let's see... yep! The Blackfire PHP extension is here.

Tip

If you have XDebug installed, disable it for the best results.

Installing the Browser Extension

At this point, our *server* is set up and ready to profile! Victory! The only thing we need *now* is a way to tell the probe when to *activate*. That's the job of the browser extension.

Go almost *all* the way back to the top of the install page where they talk about the different pieces. I'm using Chrome, so I'll click the Google Chrome extension link. I don't have it installed yet, so let's fix that: Add to Chrome.

There it is! If you refresh the docs... yep! It sees the extension.

Profiling our First Page

Hey! We're ready to profile! Ahhhh! Where should we start? Let's... just click to view details about any Big Foot sighting. All of this data comes from some data fixtures that we used to pre-populate the database while setting up the project. It uses a bunch of random data up here... and each sighting has a bunch of random comments.

When we loaded this page a second ago, the PHP extension - the probe - did nothing. To activate it, click the browser extension.

Moment of truth! When we click profile, the plugin will send a request to this page with a special header that tells the probe to activate and start profiling. Click "Profile"!

There it goes! It goes from 0 to 100% as it actually makes *10* requests and averages their data. We can also give this "profile" a name to keep our account organized: I'll say [Recording] Show page initial and hit enter.

Troubleshooting Failure

If you got to 100%, congrats! If you got an error... wah wah. This is *the* most common place for something to go wrong... and the error will almost always be the same: Probe not found. This might mean that you forgot to install the PHP extension, or that the PHP extension was installed on a different PHP binary... or that the agent isn't running... or that the agent *is* running but you misconfigured the server id and token. They have great docs to help with this.

But we had success! Click the "View Call Graph" button to go to a URL on their site. Hello beautiful Blackfire profile. Wow.

Next, let's start diving into this *mountain* of information and see how we can use it to find hidden sasquatch... I mean, hidden performance bugs.

Chapter 4: Wall Time, Exclusive Time & Other Wonders

We just made Blackfire profile our first page. One of the *best* things about Blackfire is that, instead of just... giving me some raw data-dump and saying:

> Good luck navigating *that* black pit of data!

... they expose this *treasure trove* of info on their site with a beautiful interface. This is called the "call graph". The most challenging part of Blackfire for me was learning what all this stuff means... so I could *really* get the most out of it. If you stick with me for the next few minutes, your profiling game will get a *huge* boost.

By the way, throughout the tutorial, I'll give you links to view the *exact* profile on Blackfire that I'm navigating in the video. Feel free to open it up and play around. The first one is here: https://bit.ly/sfcasts-bf-profile1.

And yes, I know, the cool-looking graph in the middle is *calling* to us, but let's start by looking at the the left side: the list of function calls, ordered from the functions that took the longest to execute on top... down to the quickest on the bottom. Well actually, Blackfire "prunes" or "removes" function calls that took *very* little time... so you won't see *everything* here.

Viewing by Different Dimensions

The functions are ordered by "time" because we're viewing the call graph in the time "dimension". You can also look at all of this information ordered by several other dimensions - like which functions took the most *memory*. It's kind of like the process manager on your computer: you can see which applications are currently taking up the most CPU, the most memory, reading the most info from your disk or even using the most network. But more on these dimensions later.

Wall Time

In the profiling world, time is called "wall time". But, it's *nothing* fancy: wall time is the difference between the time at which a function was entered and the time at which the function was left. So... wall time is a fancy word for... um... time: the amount of "time" a function took to run.

Inclusive vs Exclusive

So... we just find the function with the highest wall time and optimize it, right? Well... what if a function is taking a really long time... but actually, 99% of that time is due to a function that *it* calls. In that case, the *other* function is probably the problem.

To help sort this all out, wall time is divided into two parts: *exclusive* time and *inclusive* time. If you hover over the red graph, you'll see this: exclusive time 37.9 milliseconds, inclusive time 101 milliseconds.

Inclusive time is the *full* time it took for the function to execute. Exclusive time is more interesting: it's the time a function took to execute *excluding* the time spent inside *other* functions it called: it's a *pure* measurement of the time that the code inside *this* function took.

Right now, we're *actually* ordering this list by *exclusive* time, because that usually shows you the biggest problems. You can also order by inclusive time... which is probably not very useful: the top item is where our script starts executing, the second is the next function call, and so on. Go back to exclusive.

Navigating What Calls What

So apparently, the biggest problem, according to exclusive time, is this UnitOfWork::createEntity function... whatever that is. If you use Doctrine, you might know what this is - but let's pretend we have *no* idea.

Before we dive further into the root cause behind this slow function, the *other* way to order the calls is by the *number* of times each is called. Wow! Apparently the function that's called the most times - over 6 *thousand* times - is ReflectionProperty::setValue. Huh. I wonder who calls that?

Deeper Function Details

Click to expand that function. I love this! Even though we're viewing the call graph in the "time" dimension, this gives us *all* the info about this function: the wall time, I/O wait time, CPU time, memory footprint and network.

Wall Time = I/O Time + CPU Time

This isn't a particularly *time* consuming function - its wall time is 9.13 milliseconds. Wall time itself is broken down into two pieces, and this is important: wall time = I/O time + CPU time. There is nothing else: either a function is using CPU or it's doing some I/O operation, like talking to the filesystem or making network calls. In this case, the 9.13 milliseconds wall time is *all* CPU time.

Finding Callers

Okay, but who actually *calls* this function so many times? Above this, see those 3 down arrow buttons? These represent the *three* other functions that call this one - the size is relative to how many *times* each calls this. Click the first one. Ah ha! It's UnitOfWork::createEntity! That's the function with the highest exclusive time - it calls this function 4,959 times. Wow. So... it's definitely a problem.

If you click the other two arrows, you can see the other two callers: one calls this 984 times and the other 216 times. Both are from Doctrine.

Viewing Callees

Close all of this up and go back to ordering by the highest exclusive time. Open up UnitOfWork::createEntity(). As I mentioned, even though we're currently viewing the call graph in the "time" dimension, we can see *all* this function's dimensions right here.

Hover over the time graph: even though the exclusive time is significant - 37.9 milliseconds - most of this function's time is still *inclusive*: it's taken up by other functions that *it* calls. That helps give us a hint as to if the problem is *inside* this function... or inside something it calls.

And actually, *every* dimension has inclusive and exclusive measurements: like CPU time and even memory. If any of these had a high *inclusive* value - meaning some function *it* calls is *really* taking up that resource - you can see what functions it calls by clicking one of the arrow buttons *below* this.

What I *really* want to know though is... what's happening in our code to cause *this* function - UnitOfWork::createEntity() - be called so many times? Click the biggest arrow above. Ah: ObjectHydrator::getEntity() is the main culprit.

But... honestly... I don't know what that function is either: this is still *way* too low-level in Doctrine - I have no idea what's really going on. So next, let's use the call graph - the pretty diagram on the right - to get a full picture of what's happening going on... and how to fix it.

Chapter 5: Finding Issues via the Call Graph

There are two different ways to optimize any function: either optimize the code *inside* that function *or* you can try to call the function less times. In our case, we found that the most problematic function is UnitOfWork::createEntity. But this is a *vendor* function: it's not *our* code. So it's not something that we can optimize. And honestly, it's probably already super-optimized anyways.

But we *could* try to call it less times... if we can understand *what* in our app is causing so many calls! The call graph - the big diagram in the center of this page - holds the answer.

Call Graph: Visual Function List

Start by clicking on the magnifying glass next to createEntity. Woh! That zoomed us straight to that "node" on the right. Let's zoom out a little.

The first thing to notice is that the call graph is basically a visual representation of the information from the function list. On the left, it says this function has two "callers". On the right, we can see those two callers. But when you're trying to figure out the *big* picture of what's going on, the call graph is *way* nicer.

The Critical Path

Let's zoom out a bunch further. Now we can see a clear red path... that eventually leads to the dark red node down here. This is called the critical path. One of Blackfire's main jobs is to help us make sense out of all this data. One way it does that is exactly this: by highlighting the "path" to the biggest problem in our app.

I'm going to hit this little "home" icon - that will reset the call graph, instead of centering it around the createEntity node. In this view, Blackfire *does* hide some less-important information around the createEntity node, but it gives us the best overall summary of what's going on: we can clearly see the critical path. The critical thing to understand is: why is that path in our app so slow?

Let's trace up from the problem node... to find where *our* code starts. Ah, here's our controller being rendered... and then it renders a template. That's interesting: it means the problem is coming from *inside* a template... from inside the body block apparently. Then it jumps to a Twig extension called getUserActivityText()... that calls something else CommentHelper::countRecentCommentsForUser(). That's the last function before it jumps into Doctrine.

Finding the Problem

So the problem in our code is something around this getUserActivityText() stuff. Let's open up this template: main/sighting_show.html.twig - at templates/main/sighting_show.html.twig.

If you look at the site itself, each commenter has a label next to them - like "hobbyist" or "bigfoot fanatic" - that tells us how *active* they are in the great and noble quest of finding BigFoot. Over in the Twig template, we *get* this text via a custom Twig filter called user_activity_text:

42 lines templates/main/sighting_show.html.twig

```
... lines 1 - 4
{% block body %}
<div class="col">
... lines 7 - 19
{% for comment in sighting.comments %}
<div class="comment-container mb-3">
<div class="row">
... lines 23 - 25
<div class="col">
... line 27
<span>({{ comment.owner|user_activity_text }})</span>
... lines 29 - 33
</div>
</div>
</div>
{% endfor %}
</div>
{% endblock %}
... lines 41 - 42
```

If you're not familiar with Twig, no problem. The important piece is that whenever this filter code is hit, a function inside src/Twig/AppExtension.php is called... it's this getUserActivityText() method:

46 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 26
public function getUserActivityText(User $user): string
{
$commentCount = $this->commentHelper->countRecentCommentsForUser($user);
if ($commentCount > 50) {
return 'bigfoot fanatic';
}
if ($commentCount > 30) {
return 'believer';
}
if ($commentCount > 20) {
return 'hobbyist';
}
return 'skeptic';
}
}
```

This counts how many "recent" comments this user has made... and via our complex & proprietary algorithm, it prints the correct label.

Back over in Blackfire, it told us that the last call before Doctrine was CommentHelper::countRecentCommentsForUser() - that's *this* function call right here!

46 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 26
public function getUserActivityText(User $user): string
{
$commentCount = $this->commentHelper->countRecentCommentsForUser($user);
... lines 30 - 43
}
}
```

Let's go open that up - it's in the src/Service directory:

23 lines src/Service/CommentHelper.php

```
... lines 1 - 6
class CommentHelper
{
public function countRecentCommentsForUser(User $user): int
{
$comments = $user->getComments();
$commentCount = 0;
$recentDate = new \DateTimeImmutable('-3 months');
foreach ($comments as $comment) {
if ($comment->getCreatedAt() > $recentDate) {
$commentCount++;
}
}
return $commentCount;
}
}
```

Ah. If you don't use Doctrine, you might not see the problem - but it's one that can easily happen no matter *how* you talk to a database. Hold Command or Ctrl and click the getComments() method to jump inside:

208 lines src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface
{
... lines 18 - 50
/**
* @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="owner")
*/
private $comments;
... lines 55 - 187
/**
* @return Collection|Comment[]
*/
public function getComments(): Collection
{
return $this->comments;
}
... lines 195 - 206
}
```

Here's the story: each User on our site has a database relationship to the comment table: every user can have many comments. The way our code is written, Doctrine is querying for *all* the data for *every* comment that a User has *ever* made... simply to then loop over them, and count how many were created within the last 3 months:

23 lines src/Service/CommentHelper.php

```
... lines 1 - 6
class CommentHelper
{
public function countRecentCommentsForUser(User $user): int
{
$comments = $user->getComments();
$commentCount = 0;
$recentDate = new \DateTimeImmutable('-3 months');
foreach ($comments as $comment) {
if ($comment->getCreatedAt() > $recentDate) {
$commentCount++;
}
}
return $commentCount;
}
}
```

It's a massively inefficient way to get a simple count. *This* is problem number one.

It seems obvious now that I'm looking at it. But the nice thing is that... it's not a huge deal that I did this wrong originally - Blackfire points it out. And not over-obsessing about performance during development helps prevent premature optimization.

Attempting the Performance Bug Fix

So let's fix this performance bug. Open up src/Repository/CommentRepository.php. I've already created a function that will use a direct COUNT query to get the number of recent comments *since* a certain date:

64 lines src/Repository/CommentRepository.php

```
... lines 1 - 7
use App\Entity\User;
... lines 9 - 15
class CommentRepository extends ServiceEntityRepository
{
... lines 18 - 22
public function countForUser(User $user, \DateTimeImmutable $sinceDate): int
{
return (int) $this->createQueryBuilder('comment')
->select('COUNT(comment.id)')
->andWhere('comment.owner = :user')
->andWhere('comment.createdAt >= :sinceDate')
->setParameter('user', $user)
->setParameter('sinceDate', $sinceDate)
->getQuery()
->getSingleScalarResult();
}
... lines 34 - 62
}
```

Let's use this... instead of my current, crazy logic.

To access CommentRepository inside CommentHelper - this *is* a bit specific to Symfony - create a public function __construct() and *autowire* it by adding a CommentRepository $commentRepository argument:

23 lines src/Service/CommentHelper.php

```
... lines 1 - 5
use App\Repository\CommentRepository;
class CommentHelper
{
... lines 10 - 11
public function __construct(CommentRepository $commentRepository)
{
... line 14
}
... lines 16 - 21
}
```

Add a private $commentRepository property... and set it in the constructor: $this->commentRepository = $commentRepository:

23 lines src/Service/CommentHelper.php

```
... lines 1 - 5
use App\Repository\CommentRepository;
class CommentHelper
{
private $commentRepository;
public function __construct(CommentRepository $commentRepository)
{
$this->commentRepository = $commentRepository;
}
... lines 16 - 21
}
```

Now... I don't need *any* of this logic. Just return $this->commentRepository->countForUser(). Pass this $user... and go steal the DateTimeImmutable from below and use that for the second argument. Celebrate by killing the rest of the code:

23 lines src/Service/CommentHelper.php

```
... lines 1 - 7
class CommentHelper
{
... lines 10 - 16
public function countRecentCommentsForUser(User $user): int
{
return $this->commentRepository
->countForUser($user, new \DateTimeImmutable('-3 months'));
}
}
```

If we've done a good job, we will hopefully be calling that UnitOfWork function *many* less times - the 23 calls into Doctrine from CommentHelper eventually caused many, many things to be called.

So... let's profile this and see the result! We'll do that next and use Blackfire's "comparison" feature to *prove* that this change *was* good... except for one small surprise.

Chapter 6: Comparisons: Validate Performance Changes, Find Side Effects

We've just updated our code to make a COUNT query instead of querying for *all* the comments for a user... just to count them. So, the page will *definitely* be faster. Right? Are you *absolutely* sure? Well, *I* think it will be faster... but sometimes making one part of your code faster... will make other parts slower. Fortunately, Blackfire has a special way to *prove* that a performance tweak *does* in fact help.

Let's profile the page now - I'll refresh... then click to profile. Give it a name to stay organized [Recording] Show page after count query.

Ok! Let's go see the call graph! https://bit.ly/sfcast-bf-profile2

Hey! 270 milliseconds total time - the last one was 415. So it *is* faster. We win! Tutorial over!

Well... yeah, I agree: it does look faster. But an important aspect of optimization is understanding *why* something is faster. Like, did this reduce CPU time? I/O wait time? And, maybe more importantly, did this change cause anything to be *worse*? For example, a change might decrease CPU time, but *increase* memory. If that happened, would the change *really* be a good one? It depends.

Comparing Profiles

This leads me to one of my *favorite* tools from Blackfire: the ability to *compare* profiles. Click back to your dashboard: the top two profiles are from the initial profile and then the page after using the COUNT query. On the right, hover over the "Compare" button on the original, click, then click the new one.

Say hello to the *comparison* view: https://bit.ly/sf-bf-compare1-2. Everything that's faster, or "better" is in blue. Anything that's slower or worse will be in red. And yea, it looks like the new profile is better in *every* single category. Ok, the I/O wait is higher - but .1 millisecond higher - that's just "noise".

Anyways, the comparison *proves* that this *was* a good change. Really, it's a huge win! On the call graph, in the darkest blue, the critical path *this* time is the path that *improved* the most. Click the UnitOfWork call now. Wow. The inclusive time is down by 90 milliseconds and even the memory plummeted: down 1.39 megabytes.

Tip

The SQL Query information requires a Profiler plan or higher.

But wait. One of the items on top is called "SQL Queries". The total query *time* is less than before... but we've *added* 5 *more* queries. We removed these 18 queries... but added 23 new ones.

Is that a problem? Probably not. Overall, this change was good. And if having too many queries *does* create a *real* problem - not just an imaginary one of "too many queries" - Blackfire will help us discover that. The big takeaway here is: don't just assume that a performance enhancement... *is* actually better. We'll see this later - not *every* change we'll do in this tutorial will prove to be a good one.

Next: Blackfire has a deep understanding of PHP, database queries, Redis calls and even libraries like Symfony, Doctrine, Magento, Composer, eZ platform, Wordpress and others. Thanks to that, it automatically detects problems and recommends solutions.

Chapter 7: Recommendations

Head back to the Blackfire dashboard... and click into the latest profile - the one with our COUNT query improvement - https://bit.ly/sfcast-bf-profile2.

The critical path is now much less clear... there are kind of two critical paths... but neither end in a node with a red background... which would indicate an obvious issue. This might mean that there aren't any more easy performance "wins" on this page... it might be fast enough!

Focus in Improvement, Not Absolute Time

The response time from the profile was 270 milliseconds. If you're not satisfied with that, remember two things. First, we're profiling Symfony in its dev environment. Switching to prod would be faster... we'll do that soon. And second, the time you see in a profile will *never* be quite as fast as the real thing, because when the probe is activated - the PHP extension that does all the profiling - it slows things down. So don't obsess over any *absolute* numbers. Instead, focus on finding ways to *improve* each number.

Switching to Symfony's prod Environment

The function that takes up the most *exclusive* time is from something called DebugClassLoader. Ah. Our local Symfony app is currently running in its dev environment, which adds a lot of debugging tools, like the web debug toolbar. That stuff also slows things down... which makes profiling less useful: the profiler is cluttered up with function calls that won't *really* be there in production. That extra noise makes finding the *true* performance issues harder.

So, let's switch our app to the prod environment while profiling.

Open up .env, find the APP_ENV variable, and change it to prod:

29 lines .env

```
... lines 1 - 16
APP_ENV=prod
... lines 18 - 29
```

That will make things more realistic... but it also means that after... pretty much *any* change to our code, we will need to clear & warm the cache. No big deal: at your terminal, run:

php bin/console cache:clear

and then:

php bin/console cache:warmup

Ok, let's profile again! I'll refresh... just to make sure the page is working and... profile! I'll call this one [Recording] Show page in prod mode. Cool! 106 milliseconds is a huge improvement! Click to open the call graph: https://bit.ly/sf-bf-profile3

Now the function list and the call graph look a bit more useful. There's no *super* problematic, red-background node on the graph... but the function that takes up the most exclusive time - PDOStatement::execute() - at least makes sense: that's executing database queries.

Hello Recommendations

Tip

The Recommendations information requires a "Profiler" plan level or higher.

Back on our site, you *may* have noticed that each time we've profiled, a little exclamation icon showed up. If you clicked that, it would take you to a "Recommendations" section of the profile. The exclamation point was telling us that we're failing one or more Blackfire recommendations.

I dig this feature. Because Blackfire is written *for* PHP, it has special knowledge of how queries are made, how Composer works, Symfony, Magento and so many other things. The Blackfire team has *used* that knowledge to add a bunch of things that they call "recommendations". I call them "sanity checks".

For example, Blackfire counted our queries and said:

> Hey! FYI - you've got a bunch of queries on this page... maybe you should try to have less than 10.

Yea, our 43 queries *is* pretty high. Does that mean we should immediately run into our code and fix it? Nah. It's just a good thing to keep on your radar.

There's also a recommendation that Doctrine annotation metadata should be cached in production. Honestly... I'm not sure why that's there - Symfony apps come with a config/packages/prod/doctrine.yaml file that takes care of caching these when you're in the prod environment. When I tried to reproduce this later... it went away. So let's ignore it for now. If it comes back *later* when we deploy to production, then I *will* want to look into it further.

Composer Autoloader Recommendation

The *last* recommendation is *awesome*:

> The Composer autoloader class map should be dumped in production

By the way. if you don't know what something means, the cute question mark can help.

Look back at the function list: the *second* highest function *was* something related to Composer's autoload system. Blackfire *nailed* that this is an issue.

You may already know this, but when you deploy, you're *supposed* to run a special command - or add a special option - that tells Composer to dump an *optimized* autoload file. Blackfire is telling us that we forgot to do this locally.

Let's fix this: it will help clean up even *more* stuff on the profile. At your terminal, run:

```
composer dump-autoload --optimize
```

Perfect! Refresh the page... it works... and create another profile - I'll call this: [Recording] Show page after optimized autoloader. Click to view the call graph: https://bit.ly/sf-bf-profile4 and close the old one.

It's not *significantly* faster, but we've removed at least one heavy-looking function call from our list. That will help us focus on any *real* problems. Check out the recommendations now. Yea! The Composer one is *gone*. Later, we'll learn how to add custom assertions - which are basically a way to write your *own* custom recommendations.

Next, let's look deeper at what's going on with this PDOStatement::execute stuff. Is our page fast enough? Or can we discover some further, hidden optimizations?

Chapter 8: Property Caching

Now that we've got our application in production mode and we've dumped the autoloader, it's easier to see what the biggest performance problem is on this page: https://bit.ly/sf-bf-profile4

And actually, there might *not* be any more problems worth solving. I mean, it's loading in 104 milliseconds... *even* with the Probe doing all the profiling work.

But... let's see for sure. The function with the highest exclusive time *now* is PDOStatement::execute()... which is a low-level function that *executes* SQL queries.

Tip

The SQL Query information requires a Profiler plan or higher.

If we hover over the query info, these are only taking 12.5 milliseconds... but we *are* making 43 SQL calls on this page. Is that a problem? It's not ideal, but is it worth fixing? I guess it depends on how much you care... and whether the fix would be easy or if it would add a lot of complexity to our app.

Navigating the Call Graph: Top to Bottom, Bottom to Top

When you're trying to identify where the problem is, there are two ways to look at the call graph - and I often do both to help me understand what's going on. First, you can read from top to bottom - trace through your *whole* application flow to figure out what's going on down the hot path. Or, you can do the opposite: start at the bottom - start *where* the problem is... and trace up to find where your code starts.

Let's start from the top: handleRaw() is the framework booting up... and as we trace down... it renders our controller, renders our template... and we're once again inside the body block. This is really the same as last time! Our AppExtension::getUserActivityText() calls the countRecentCommentsForUser() function 23 times. That makes sense: we probably have 23 comments on the page... and for each comment, we need to count *all* the author's comments to print out this label.

Navigating Dimensions

Before we think about if, and *how* we might fix this, let's back up and look at other *dimensions* of this profile. In addition to wall time, we can completely re-draw the call graph based on only I/O time or CPU time. Remember, wall time is I/O time + CPU time. Or we could do something totally different: look at which functions are using the most *memory*... or even the most network bandwidth.

When we look at this in the network dimension, PDOStatement::execute() - the function that makes SQL calls - shows up here as a *big* problem. That's because SQL queries are technically network requests.

Re-draw the call graph for the I/O Wait time dimension. We see the same problem here because network calls - and so SQL calls - are part of I/O wait time.

The point is: while "wall time" is *typically* the most useful dimension, don't forget about these other ones: they can give us more information about what's going on. Is a function slow because of inefficient code inside? Or is it, for example, because of a network call?

Click back to I/O wait time - PDOStatement::execute() is *definitely* the issue according to this - and the critical path is pretty clear. This *one* function is taking over *half* the I/O wait time... but that's only 6 milliseconds. Optimizing this might not be worth it... but let's at least see if we can figure out how to call it less times.

As we already discovered, the problem is coming from CommentRepository::countForUser() which is called by AppExtension::getUserActivityText().

Over in src/Twig/AppExtension.php, each time we render a comment, it calls countForUser() and passes the User object attached to this comment:

46 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 26
public function getUserActivityText(User $user): string
{
$commentCount = $this->commentHelper->countRecentCommentsForUser($user);
... lines 30 - 43
}
}
```

Property Caching

Can we optimize this? Well... sometimes, the *same* user will comment many times on the same sighting - like this vborer user. When that happens, we're making a query to count that user's comments for *every* comment. That's wasteful!

So here's one idea: leverage "property caching". Basically, we'll keep track of the "status" strings for each user and use that to avoid calculating the status more than once for a given user.

Start by moving most of the logic into a private function called calculateUserActivityText(): this will have a User argument and return a string:

57 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 37
private function calculateUserActivityText(User $user): string
{
$commentCount = $this->commentHelper->countRecentCommentsForUser($user);
if ($commentCount > 50) {
return 'bigfoot fanatic';
}
if ($commentCount > 30) {
return 'believer';
}
if ($commentCount > 20) {
return 'hobbyist';
}
return 'skeptic';
}
}
```

Next, add a new property to the top of the file: private $userStatuses = []:

57 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 14
private $userStatuses = [];
... lines 16 - 55
}
```

Back in the public function, here's the magic: if *not* isset($this->userStatuses[$user->getId()]), then set it by saying $this->userStatuses[$user->getId()] = $this->calculateUserActivityText($user). At the bottom of the function, return $this->userStatuses[$user->getId()]:

57 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 28
public function getUserActivityText(User $user): string
{
if (!isset($this->userStatuses[$user->getId()])) {
$this->userStatuses[$user->getId()] = $this->calculateUserActivityText($user);
}
return $this->userStatuses[$user->getId()];
}
... lines 37 - 55
}
```

This is one of my *favorite* performance tricks because it has *no* downside, except for some extra code. If getUserActivityText() is called and passed the same User multiple times within a single request, we won't duplicate any work.

So... we probably made our site faster, right? Let's find out! Since we're in Symfony's prod environment, just to be safe, let's clear the cache:

php bin/console cache:clear

and warm it up:

php bin/console cache:warmup

Back in the browser, refresh the page and... let's profile! I'll name this one [Recording] show page try property caching. View the call graph: https://bit.ly/sf-bf-profile-prop-caching.

Ok - PDOStatement still looks like a main problem... but I think we're a *little* faster. You know what? Let's just compare the two profiles. Go back to the dashboard and compare the previous profile to this one. https://bit.ly/sf-bf-compare-prop-caching. I'll close the old profile.

Ok, so it *did* help - lower time in each dimension... and we saved 5 queries. So, this is a win, right? *Maybe*. If you profiled other Big foot sighting pages, which I did, you would find that this often did *not* help... or helped *very* little. In fact, this is the *first* time I've seen it help *nearly* this much.

So, does the improvement justify the added complexity in our code? If we can repeat this 13% improvement consistently, yea, it is. But if it's more like 1%, probably not.

And even 13% is not *that* much... and PDOStatement::execute() is *still* the biggest problem. I feel like the profile is trying to ask us: is there a *better* way to optimize this?

Next, let's try another approach: using a *real* cache layer. *Truly* caching things has its own downside: added complexity in your code and *possibly* - depending on what you're caching - the need to worry about *invalidating* cache. We'll want to be sure it's worth it.

Chapter 9: Using a Caching Layer & Proving its Worth

Whenever we make something more performant, we often *also* make our code more complex. So, was the property-caching trick we just used worth it? Maybe... but I'm going to revert it.

Remove the property caching logic and just return $this->calculateUserActivityText($user):

51 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 26
public function getUserActivityText(User $user): string
{
return $this->calculateUserActivityText($user);
}
... lines 31 - 49
}
```

And... we don't need the $userStatuses property anymore:

57 lines src/Twig/AppExtension.php

```
... lines 1 - 10
class AppExtension extends AbstractExtension
{
... lines 13 - 14
private $userStatuses = [];
... lines 16 - 55
}
```

We *could* stop here and say: this spot is not worth optimizing. Or, we can try a different solution - like using a *real* caching layer. After all, this label probably won't change very often... and it's probably not *critical* that the label changes at the *exact* moment a user adds enough comments to get to the next level. Caching could be an easy win.

Adding Caching

Back in AppExtension, autowire Symfony's cache object by adding an argument type-hinted with CacheInterface - the one from Symfony\Contracts\Cache. I'll press Alt+Enter and select "Initialize fields" to make PhpStorm create a new property with this name and set it in the constructor:

61 lines src/Twig/AppExtension.php

```
... lines 1 - 7
use Symfony\Contracts\Cache\CacheInterface;
... lines 9 - 12
class AppExtension extends AbstractExtension
{
... line 15
private $cache;
public function __construct(CommentHelper $commentHelper, CacheInterface $cache)
{
... line 20
$this->cache = $cache;
}
... lines 23 - 59
}
```

Down in the method, let's first create a cache key that's specific to each user. How about: $key = sprintf('user_activity_text_'.and then $user->getId():

61 lines src/Twig/AppExtension.php

```
... lines 1 - 12
class AppExtension extends AbstractExtension
{
... lines 15 - 30
public function getUserActivityText(User $user): string
{
$key = sprintf('user_activity_text_'.$user->getId());
... lines 34 - 39
}
... lines 41 - 59
}
```

Wow, I *just* realized that my sprintf here is totally pointless.

Then, return $this->cache->get() and pass this $key. If that item exists in the cache, it will return immediately:

61 lines src/Twig/AppExtension.php

```
... lines 1 - 12
class AppExtension extends AbstractExtension
{
... lines 15 - 30
public function getUserActivityText(User $user): string
{
$key = sprintf('user_activity_text_'.$user->getId());
return $this->cache->get($key, function(CacheItemInterface $item) use ($user) {
... lines 36 - 38
});
}
... lines 41 - 59
}
```

Otherwise, it will execute this callback function, pass us a CacheItemInterface object and *our* job will be to return the value that *should* be stored in cache.

Hmm... I need the $user object inside here. Add use then $user to bring it into scope. Then return $this->calculateUserActivityText($user):

61 lines src/Twig/AppExtension.php

```
... lines 1 - 12
class AppExtension extends AbstractExtension
{
... lines 15 - 30
public function getUserActivityText(User $user): string
{
$key = sprintf('user_activity_text_'.$user->getId());
return $this->cache->get($key, function(CacheItemInterface $item) use ($user) {
... lines 36 - 37
return $this->calculateUserActivityText($user);
});
}
... lines 41 - 59
}
```

I think it's probably safe to cache this value for one hour: that's long enough, but not *so* long that we need to worry about adding a system to manually *invalidate* the cache. Set the expiration with $item->expiresAfter(3600):

61 lines src/Twig/AppExtension.php

```
... lines 1 - 12
class AppExtension extends AbstractExtension
{
... lines 15 - 30
public function getUserActivityText(User $user): string
{
$key = sprintf('user_activity_text_'.$user->getId());
return $this->cache->get($key, function(CacheItemInterface $item) use ($user) {
$item->expiresAfter(3600);
return $this->calculateUserActivityText($user);
});
}
... lines 41 - 59
}
```

So... does this help? Of course it will! More importantly, because we decided we don't need to worry about adding more complexity to *invalidate* the cache, it's probably a big win! But let's find out for sure.

Move over and refresh. Boo - 500 error. We're in the prod environment... and I forgot to rebuild the cache:

php bin/console cache:clear

And:

php bin/console cache:warmup

Profiling with Cache

Refresh again. And... profile! I'll name this one: [Recording] Show page real cache. Open up the call graph: https://bit.ly/sf-bf-real-caching.

This time things look *way* better. But let's not trust it: go compare the *original* profile - before we even did property caching - to this new one: https://bit.ly/sf-bf-compare-real-cache.

Wow. The changes are significant... and there's basically no downside to the changes we made. Even our memory went down! You can also compare this to the property caching method: https://bit.ly/sf-bf-compare-prop-real-caching. Yea... it's way better

And really, this is *no* surprise: *fully* caching things will... of course be faster! The *question* is how *much* faster? And if adding caching means that you *also* need to add a cache invalidation system, is that performance boost worth it? Since we don't need to worry about invalidation in this case, it was *totally* worth it.

Next: let's find & solve a classic N+1 query problem. The final solution might not be what you traditionally expect.

Chapter 10: The N+1 Problem & EXTRA_LAZY

At this point, I'm pretty happy with the show page that we've been profiling. So let's look at something different: let's profile the homepage at https://localhost:8000/.

Ok, this page has a list of all of the sightings... and on the right, that shows some SymfonyCasts repository info from GitHub. Let's refresh... though... that's not really needed - and profile! I'll call this one: [Recording] Original homepage - https://bit.ly/sf-bf-homepage-original.

Ok! 165 milliseconds! Let's view the call graph. Well... this looks familiar! We have the *same* number 1 exclusive-time function as before: UnitOfWork::createEntity(). In *that* situation, it meant that we were querying for too many items and so Doctrine was *hydrating* too many objects. Is it the same problem now? And if so, why? Can we optimize it?

Time to put on our profiling detective hats. Let's follow the hot path! We enter MainController::homepage() and render a template... so the problem is coming from our *template*. Interesting. Next _sightings.html.twig is rendered... and then something called twig_length_filter executes loadOneToManyCollection(), which is from Doctrine. Let's do some digging in that template: templates/main/_sightings.html.twig.

We saw that it was referencing something called twig_length_filter. Search the template for length. Ah: sighting.comments|length:

16 lines templates/main/_sightings.html.twig

```
{% for sighting in sightings %}
<tr>
... lines 3 - 10
<td>
<a class="text-white table-content text-center" href="{{ path('app_sighting_show', {id: sighting.id}) }}">{{ sighting.comments|length }}</a>
</td>
</tr>
{% endfor %}
```

Finding the N+1 Problem

Look back on the site: one of the things it does is prints the number of *comments* for each article. The length filter counts how many items are in sighting.comments, which is a database relationship from the big_foot_sighting table to the comment table.

If you're not familiar with Doctrine, when you call sighting.comments, at that moment, Doctrine queries for *all* of the comments for that specific BigFootSighting record. I'll open up src/Entity/BigFootSighting.php. Yep, we're accessing the comments property, which is a OneToMany relationship to Comment:

207 lines src/Entity/BigFootSighting.php

```
... lines 1 - 11
class BigFootSighting
{
... lines 14 - 56
/**
* @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="bigFootSighting")
* @ORM\OrderBy({"createdAt"="DESC"})
*/
private $comments;
... lines 62 - 205
}
```

The point is: for *each* BigFootSighting that we are rendering, Doctrine is making an *extra* query to fetch *all* the comments for that sighting. This is basically the classic N+1 problem. If we want to print 25 BigFootSighting rows, in addition to the 1 query to fetch the 25 rows, the system will *also* make 25 *additional* queries to fetch the *comments* for each sighting. That's 25 + 1 queries.

You can see this in the SQL queries in Blackfire: we have one query from big_foot_sighting - the query above is related to the pagination logic - then *25* queries from the comment table.

Counting with fetch="EXTRA_LAZY"

Okay, we have identified the problem: we are not only making a lot of queries... but those queries are *also* fetching *all* the comment data... just to count them. Silliness!

One *simple* solution might be... just to tell Doctrine to make a COUNT query instead of fetching all the data. We would *still* have 25 extra queries... but they would be much faster.

In Doctrine, we can do this really easily. If you access a relationship - like the comments property - and *only* count it, we can *ask* Doctrine to do a COUNT query instead of loading *all* the comment data. How? Above the comments property, add fetch="EXTRA_LAZY":

207 lines src/Entity/BigFootSighting.php

```
... lines 1 - 11
class BigFootSighting
{
... lines 14 - 56
/**
* @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="bigFootSighting", fetch="EXTRA_LAZY")
... line 59
*/
private $comments;
... lines 62 - 205
}
```

Before we try this, don't forget that we're in the prod environment: run cache:clear:

php bin/console cache:clear

And cache:warmup:

php bin/console cache:warmup

Ok, let's see if this helps! Spin over, refresh the page and... profile! I'll call this one: [Recording] homepage EXTRA_LAZY - https://bit.ly/sf-bf-extra-lazy. I'll close the other tab and view the call graph.

Was this better? Well, createEntity() isn't the biggest problem anymore... so that's a good sign! Let's compare to be sure: go from the original homepage... to the most recent profile: https://bit.ly/sf-bf-extra-lazy-compare.

And... wow! Yea, this is a *huge* win in every category! So, was this a good change? Absolutely: this was an *awesome* change.

But, even though the queries are much faster... we're still making the same *number* of queries. Is that something we care about? I don't know? But that's the great thing about profiling with Blackfire: you don't need to *absolutely* optimize everything. If you're not sure if something is a problem, you can deploy and check it on production to see if it's *really* slowing things down under realistic conditions. *Especially* because sometimes improving performance comes at a cost of extra complexity.

Next, let's see if we *can* reduce the number of queries. Will it help performance? If so, is it enough for the added complexity?

Chapter 11: Fixing N+1 With a Join?

We made a *huge* leap forward by telling Doctrine to make COUNT queries to count the comments for each BigFootSighting... instead of querying for *all* the comments *just* to count them. That's a big win.

Could we go further... and make a smarter query that can grab all this data at once? That *is* the classic solution to the N+1 problem: need the data for some Bigfoot sightings *and* their comments? Add a JOIN and get all the data at once! Let's give that a try!

Adding he JOIN

The controller for this page lives at src/Controller/MainController.php - it's the homepage() method:

122 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
/**
* @Route("/", name="app_homepage")
*/
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
$sightings = $this->createSightingsPaginator(1, $bigFootSightingRepository);
return $this->render('main/homepage.html.twig', [
'sightings' => $sightings
]);
}
... lines 30 - 120
}
```

To help make the query, this uses a function in src/Repository/BigFootSightingRepository.php - this findLatestQueryBuilder():

59 lines src/Repository/BigFootSightingRepository.php

```
... lines 1 - 15
class BigFootSightingRepository extends ServiceEntityRepository
{
... lines 18 - 22
public function findLatestQueryBuilder(int $maxResults): QueryBuilder
{
return $this->createQueryBuilder('big_foot_sighting')
->setMaxResults($maxResults)
->orderBy('big_foot_sighting.createdAt', 'DESC');
}
... lines 29 - 57
}
```

This method ... if you did some digging ... creates the query that returns these results.

And... it's fairly simple: it grabs all the records from the big_foot_sighting table, orders them by createdAt and sets a max result - a LIMIT.

To *also* get the comment data, add leftJoin() on big_foot_sighting.comments and alias that joined table as comments. Then use addSelect('comments') to not only *join*, but also *select* all the fields from comment:

61 lines src/Repository/BigFootSightingRepository.php

```
... lines 1 - 15
class BigFootSightingRepository extends ServiceEntityRepository
{
... lines 18 - 22
public function findLatestQueryBuilder(int $maxResults): QueryBuilder
{
return $this->createQueryBuilder('big_foot_sighting')
->leftJoin('big_foot_sighting.comments', 'comments')
->addSelect('comments')
->setMaxResults($maxResults)
->orderBy('big_foot_sighting.createdAt', 'DESC');
}
... lines 31 - 59
}
```

Let's... see what happens! To be safe, clear the cache:

php bin/console cache:clear

And warm it up:

php bin/console cache:warmup

Now, move over, refresh and profile! I'll call this one: [Recording] Homepage with join: https://bit.ly/sf-bf-join.

Go check it out! Woh! This... looks weird... it looks *worse*! Let's do a compare from the EXTRA_LAZY profile to the new one: https://bit.ly/sf-bf-join-compare.

Wow... this is much, much worse: CPU is way up, I/O... it's up in every category, especially network: the amount of data that went over the network. We *did* make less queries - victory! - but they took 8 milliseconds longer. We're now returning *way* more data than before.

So this was a *bad* change. It seems obvious now - but in a different situation where you might be doing different things with the data, this *same* solution could have been the right one! Let's remove the join and rely on the EXTRA_LAZY solution.

A Smarter Join?

Yes, this *will* mean that we will once again have 27 queries. If you don't like that, there *is* another solution: you could make the JOIN query smarter - it would look like this:

```
// src/Repository/BigFootSightingRepository.php
public function findLatestQueryBuilder(int $maxResults): QueryBuilder
{
    return $this->createQueryBuilder('big_foot_sighting')
        ->leftJoin('big_foot_sighting.comments', 'comments')
        ->groupBy('big_foot_sighting.id')
        ->addSelect('COUNT(comments.id) as comment_count')
        ->setMaxResults($maxResults)
        ->orderBy('big_foot_sighting.createdAt', 'DESC');
}
```

The key is that instead of selecting *all* the comment data... which we don't need... this selects *only* the count. It gets the *exact* data we need, in one query. From a performance standpoint, it's probably the perfect solution.

But... it has a downside: complexity. Instead of returning an array of BigFootSighting objects, this will return an array of... arrays... where each has a 0 key that is the BigFootSighting object and a comment_count key with the count. It's just... a bit weird to deal with. For example, the template would need to be updated to take this into account:

```
{% for sightingData in sightings %}
    {% set sighting = sightingData.0 %}
    {% set commentCount = sightingData.comment_count %}

    {# ... #}
        {{ sighting.title }}

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

And... because of the pagination that this app is using... the new query would actually produce an error. So let's keep things how they are now. If the extra queries ever become a *real* problem on production, *then* we can think about spending time improving this. Sometimes profiling is about knowing what *not* to fix... because it may not be worth it.

Next, if you were surprised that we didn't see any evidence of the network request that the homepage is making to render the SymfonyCasts repository info from GitHub, that's because the homepage is more complex than it might seem. Let's use a cool "Profile all" feature to see *all* requests that the homepage makes.

Chapter 12: Profile All Requests (Including Ajax)

When you open the browser extension to create a profile, it has a few options that we've been... ignoring so far.

Debugging Mode

Tip

Debugging mode is available via the Debugging add-on.

For example, "debugging mode" will tell Blackfire to *disable* pruning - that's when it removes data for functions that don't take a lot of resources - and also to disable anonymization - that's when it hides exact details used in SQL queries and HTTP requests. Debugging mode is nice if something weird is going on.. and you want to *fully* see what's happening inside a request.

Distributed Profiling

Tip

Distributed profiling is available to Premium plan users or higher.

Another superpower of Blackfire is called distributed profiling... which you either won't care about... or it's the most awesome thing ever. Imagine you have a micro-service architecture where, when you load the page, it makes a few HTTP requests to some microservices. If you have Blackfire installed on all of your microservices, Blackfire will automatically create profiles for *every* request made to *every* app. The final result is a profile with sub-profiles that show you how the entire infrastructure is working together. It's... pretty incredible.

But, if you want to disable it and *only* profile this main app, you can do that with this option.

Disabling Aggregation

The last option is to "disable aggregation". That's a fancy way of telling Blackfire that you want to make & profile just *one* request, instead of making 10 requests and averaging the results.

Profiling All Requests

But what I *really* want to look at is this "Profile all requests" link. Hit "Record"... then refresh. Woh! Cool! It already made 2 requests! And if I scroll down a little bit... there's a third! Let's stop right there.

That jumps us to our Blackfire dashboard. These *last* three profiles were just created: one for the homepage and two others: these are both AJAX calls! Surprise! Without even thinking about it, we discovered a few extra requests that are part of that page.

This first one - /api/github-organization - is what loads this GitHub repository info on the right. This makes an API call for the most popular repositories under the Symfonycasts organization... which is kind of silly... but it was a *great* way to show how network requests look in Blackfire. We'll see that in a minute.

This other request - for /_sightings - is an AJAX call that powers the forever scroll on the page.

Basically... I like using "profile all requests" in 3 situations. One, to get an idea of what's all happening on a page. Two, to profile AJAX requests... though I'll show you another way to do that soon. And three, to profile form submits: fill out the form, hit "Record", then submit.

Checking out the Network Requests

Let's look closer at the /api/github-organization AJAX profile: https://bit.ly/sf-bf-github-org. As I mentioned, this makes a network request to the GitHub API to load repository information. The profile... is almost comical! Out of the 438 millisecond wall time - 82% of it is from curl_multi_select() - that's the time spent making any API calls.

It's kind of fun to look at this in the CPU dimension, which is only 74 milliseconds. curl_multi_exec() is *still* the biggest offender... but it's a lot less obvious what the critical path is. Compare that with the I/O wait dimension, which includes network time. The critical path is *ridiculously* obvious here. This is an extreme example of how different dimensions can be more or less useful depending on the situation.

One of the interesting things is that... this is *not* the full call graph. According to this, the code goes straight from handleRaw() - which is the first call into the Symfony Framework - to our controller. In reality, there are many more function calls in between. Switch back to the CPU dimension. Yep! This shows more nodes.

This is the result of that "pruning" I mentioned a few minutes ago. Blackfire removes function calls that don't consume any significant resources so that the critical path - from a *performance* standpoint - is more obvious. The call graph also automatically hides or shows some info based on what you're zoomed in on.

In this situation, the critical path is obvious. You can also see the network requests on top. There are actually *two*: one that returns 1.5 kilobytes and another that returns 5.

This shows the network time too... but at *least* if you're using the Symfony HTTP client like I am, these numbers aren't right - they're far too small... I think that's due to the asynchronous nature of Symfony's HTTP Client. That's ok - because the overall cost *is* showing up correctly in all the other dimensions.

So how do we fix this? Should we add some caching? Or somehow try to make only *one* API call instead of two? We're actually going to revisit and fix this

problem later. For now, I wanted you to be aware of the "Profile All" feature. Next, let's check out the Blackfire command-line tool, which has *two* superpowers... one of which has nothing to do with the command line.

Chapter 13: The Blackfire CLI Tool for AJAX Requests

We know that the probe - that's the Blackfire PHP extension - doesn't run on every single request: it only runs when it detects that our browser extension is *telling* it to run.

There's actually a *second* way that you can tell the probe to do its work. It's with a *super* handy command-line tool.

Installing the Blackfire CLI Tool

Go back to the Blackfire site, click on their docs... and once again find the installation page. When we went through this earlier, we purposely skipped one step: installing the Blackfire CLI tool. Actually, Blackfire recently updated this page... and I like the newer version a lot better. In both versions of the docs - the new one and the old one you see here - if you followed the commands to install the "agent" then you've already *also* installed the CLI tool. Nice!

To make sure, find your terminal and try running:

blackfire version

Blackfire CLI Confiug: Client ID & Token

Got it! Before using this, we *do* need to add a little bit of configuration by running a blackfire config command. On the old version of the docs, I'll copy the "client ID": I'll need that in a second. On the newer version of the docs, you'll be able to copy a blackfire config command that already includes the client id and client token. For me, I'll run

blackfire config

If your version of the command has the --client-id and --client-token options already, you're done! If not, like me, paste in the Client Id... then also copy and paste in the token.

The client id and token work... almost like a username and password to your Blackfire account. When we use the browser extension, we're logged into Blackfire in the browser. When we click profile, the Blackfire API is able to give the extension some credentials that it passes to the probe to prove that we're allowed to profile this page.

When you use the Blackfire command line tool to profile something... the client id and client token are used to talk to the Blackfire API and get those *same* credentials that it then passes to the probe to prove we're authorized to profile. They basically identify & prove which *user* we are on Blackfire.

Profiling AJAX Requests

The Blackfire CLI tool has two superpowers. The first is that you can run blackfire curl and then pass a URL to *any* page on your site that you want to create a profile for. Now... that *might* seem *totally* worthless. After all... if we want to profile a page... isn't it easier just to *go* to that page in our browser and use the extension to profile it?

Yep! Unless... you *can't* easily "go" to that page - like if you want to profile an AJAX request or an API endpoint. Check this out: I'll open up the dev tools, go to the "Network" section and refresh. Notice I'm already filtered to XHR requests - so the /api/github-organization AJAX request pops up. Want to easily profile *just* that request? Right click on it and select "Copy as cURL".

Now head *back* to your terminal and paste. Cool, right? It creates a *full* curl command that you can use to make that same request... *including* any session cookies, which means this request will be authenticated as the same user you're logged in as in the browser. We can use this with Blackfire: say blackfire then paste!

Try it! It's profiling and using the same process as the browser: making 10 requests and profiling each one. This is my favorite way to profile AJAX requests. When it finishes, it gives us the URL to the call graph and some basic stats below. Go open that profile: http://bit.ly/sf-bf-curl!

It works! Use that to easily profile *any* AJAX requests you want to.

So what is the *second* superpower of the CLI tool? It's actually its *main* superpower: the ability to profile command-line scripts. Let's do that next.

Chapter 14: Profiling Command Line scripts

As handy as the CLI tool is for profiling AJAX requests, its *true* purpose is something different: it's to allow us to profile our custom command-line scripts. Let's check out an example. I've already created a command line script that you can execute by calling:

php bin/console app:update-sighting-scores

What does it do? Let me show you! Each Bigfoot sighting on the site has, what we call, a "Bigfoot believability score". Right now, this shows zero for *every* sighting. That's because we use a highly-complex and proprietary algorithm to calculate this. It's *such* a heavy process that, instead of figuring it out on page-load, we store the current value in a column on each row of the table. To *populate* that column, we run this command once a day: it loops over all the sightings, calculates the newest "believability score" and saves it back to the database. Try it:

php bin/console app:update-sighting-scores

It takes a few seconds... and when we go back to the site and refresh... we find out that this Bigfoot sighting in *kind of* believable - a score of 5 out of 10.

The code for this lives at src/Command/UpdateSightingScoresCommand.php:

54 lines src/Command/UpdateSightingScoresCommand.php

```
... lines 1 - 2
namespace App\Command;
use App\Repository\BigFootSightingRepository;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;
class UpdateSightingScoresCommand extends Command
{
protected static $defaultName = 'app:update-sighting-scores';
private $bigFootSightingRepository;
private $entityManager;
public function __construct(BigFootSightingRepository $bigFootSightingRepository, EntityManagerInterface $entityManager)
{
$this->bigFootSightingRepository = $bigFootSightingRepository;
$this->entityManager = $entityManager;
parent::__construct();
}
protected function configure()
{
$this
->setDescription('Update the "score" for a sighting')
;
}
protected function execute(InputInterface $input, OutputInterface $output)
{
$io = new SymfonyStyle($input, $output);
$sightings = $this->bigFootSightingRepository->findAll();
$io->progressStart(count($sightings));
foreach ($sightings as $sighting) {
$io->progressAdvance();
$characterCount = 0;
foreach ($sighting->getComments() as $comment) {
$characterCount += strlen($comment->getContent());
}
$score = ceil(min($characterCount / 500, 10));
$sighting->setScore($score);
$this->entityManager->flush();
}
$io->progressFinish();
}
}
```

You *might* already see a problem. But if you don't... that's ok! Let's see what Blackfire thinks. This time, run that *same* command, but put blackfire run at the beginning:

```
blackfire run bin/console app:update-sighting-scores
```

Woh. It's a *lot* slower now: we're seeing evidence of how the PHP extension slows down the process... and wow... it's just getting slower, and slower. I'm going to use the magic of TV to speed things up.

Ok, let's look at that profile! http://bit.ly/sf-bf-console-original

Woh! Some computeChangeSet() function was called almost 500,000 times! Ah! That's taking up *half* of the exclusive time! Because this call is *such* a problem, Blackfire is hiding a *lot* of data, all of which is unimportant relative to what we *are* seeing.

That's cool because the result is a *super* simple call graph: here's our command... here's EntityManager::flush()... and then it goes into deep Doctrine stuff.

Let's check out the command and look for the EntityManager::flush() call:

54 lines src/Command/UpdateSightingScoresCommand.php

```
... lines 1 - 11
class UpdateSightingScoresCommand extends Command
{
... lines 14 - 33
protected function execute(InputInterface $input, OutputInterface $output)
{
... lines 36 - 39
foreach ($sightings as $sighting) {
... lines 41 - 48
$this->entityManager->flush();
}
... line 51
}
}
```

Yep! I flush once each time at the end of the loop, which updates that database row. If you're familiar with Doctrine, you might know the problem: you don't *need* to call flush() inside the loop. Instead, move this *after* the loop:

54 lines src/Command/UpdateSightingScoresCommand.php

```
... lines 1 - 11
class UpdateSightingScoresCommand extends Command
{
... lines 14 - 33
protected function execute(InputInterface $input, OutputInterface $output)
{
... lines 36 - 39
foreach ($sightings as $sighting) {
... lines 41 - 48
}
$this->entityManager->flush();
... line 51
}
}
```

With this change, Doctrine will try to perform *all* update queries at the *same* time... which even lets it try to *optimize* those queries if it can. But the *big* problem with our old code was something related to Doctrine's UnitOfWork::computeChangeSet(). *Each* time you call flush() in Doctrine, it looks at *all* the objects it has queried for - so *all* of the BigFootSighting objects - and checks *every single one* to see if any data has changed that needs to be sync'ed back to the database with an UPDATE query. Yep, with the *old* code, it was checking *every* property of *every* record for updated data on *every* loop. Hence...the 450,000 calls!

Let's profile again with the updated code.

```
blackfire run php bin/console app:update-sighting-scores
```

This time it's *much* faster - I don't even think we need to compare the profiles: 56 seconds down to 1. Open it up: http://bit.ly/sf-bf-console2.

Complexity, Speed & Reliability

Could we optimize this further? Maybe! But this performance enhancement already came at a cost: reduced reliability. I originally put the call to flush() inside the loop *not* because I didn't know better... but to make the command a little more resilient. If, for example, the command gets through *half* of the records and then has an error, with the *new* code, *none* of the scores will be saved.

It's beyond the scope of this tutorial, but I *love* to make my command-line scripts *super* forgiving. If this were a real app, I would probably save the datetime that I last calculated the score for each record and use that to query for *only* the rows that have *not* been updated in the last 24 hours. I would *also* move the

flush() back into the loop:

```
$sightings = $this->bigFootSightingRepository
    ->findAllScoreNotUpdatedSince(new \DateTime('-1 month'));

foreach ($sightings as $sighting) {
    // ...

    $sighting->setScore($score);
    $sighting->setScoreLastUpdatedAt(new \DateTime());
    $this->entityManager->flush();
}
```

Thanks to those changes, if this command failed half-way through, the first half of the records would already be updated and we could run the command again to *resume* with the ones that are still *not* updated.

But wouldn't that make the command super-slow again? Yep! And with the help of Blackfire, you can test solutions that improve performance without making the command less reliable. For example, we could make the first query only return an array of integer ids. Then, inside the loop, use that id to query for the *one* object you need. That would mean we only have *one* BigFootSighting object in memory at a time instead of all of them:

```
$sightingIds = $this->bigFootSightingRepository
    ->findIdsScoreNotUpdatedSince(new \DateTime('-1 month'));

foreach ($sightingIds as $id) {
    $sighting = $this->bigFootSightingRepository->find($id);

    $sighting->setScore($score);
    $sighting->setScoreLastUpdatedAt(new \DateTime());
    $this->entityManager->flush();
}
```

You can go further by calling EntityManager::clear() after flush() to, sort of, "clear" Doctrine's memory of the BigFootSighting object you just finished... so that it doesn't check it for changes when we call flush() during the *next* time through the loop:

```
$sightingIds = $this->bigFootSightingRepository
    ->findIdsScoreNotUpdatedSince(new \DateTime('-1 month'));

foreach ($sightingIds as $id) {
    $sighting = $this->bigFootSightingRepository->find($id);

    $sighting->setScore($score);
    $sighting->setScoreLastUpdatedAt(new \DateTime());
    $this->entityManager->flush();
    $this->entityManager->clear($sighting);
}
```

The point is: like with *everything*, make your code do what it needs to... then use Blackfire to solve the real performance issues... if you have any.

Next, there's a *giant* screen in Blackfire that we haven't even looked at yet. What!? It's... the Timeline!

Chapter 15: Timeline: Go Behind-the-Scenes with your Code

Click log in to find our super-secure login system. We not only give you a valid email address, but even the password! We're *very* generous to our users.

You can't tell, but now that we're logged in, a new piece of code is... silently running in the background on each request. Blackfire is going to help us notice this.

Back to the dev Environment

Before we profile this page, open up the .env file and switch *back* to the dev environment:

29 lines .env

```
... lines 1 - 16
APP_ENV=dev
... lines 18 - 29
```

What I'm about to show you is more of a *debugging* tool than a *profiling* tool. We're switching back to the dev environment both to make our life a little bit easier - no need to clear the cache after changes - *and* because when your code executes more slowly, Blackfire tends to prune, or remove, less stuff. That's *bad* for trying to find performance issues, but *good* if your goal is to debug something... or understand how your app is working.

I'll refresh the page to make sure that it works. Yep! Our handy web debug toolbar on the bottom is back! Let's profile! I'll call this one [Recording] Homepage authenticated dev: http://bit.ly/sf-bf-timeline. Poetry.

When that finishes, as usual, click to view the call graph. Okay: there's not too much interesting here... especially because the DebugClassLoader stuff is once again adding "noise" that won't be there on production. It's not clear what the critical path is... and the page, at this point, is probably fast enough for me.

Hello Timeline

So let's look at something else: click the "Timeline" link. OooOOOOo. The timeline... other than just *looking* cool... is *the* place to go to... just... basically figure out how your app is working: how does the code flow through all the layers? What hidden things might be happening?

For example, this page apparently has 28 SQL queries. But where are these happening? Are they all in the controller? Are some in the controller and others are in the template? Are some coming from somewhere else we didn't even think of? That's something that the call graph can't really tell us.

I love the timeline... but I'll admit that the first few times I looked at this page... I didn't really understand what was going on... or how to make this useful. It *looks* simple enough - we can see the function calls and their child calls from left to right through the lifecycle of the request - but there's more to it.

Metrics

Let's start on the left: these timeline metrics. Metrics are basically a way that Blackfire groups function calls together and give them a label. For example, Blackfire knows that a *specific* function call means that an *event* is being dispatched. It finds those, labels all of them as symfony.events and give them this purple color so that they show up more clearly on the right. Here's one Symfony event right here... and there's another one.

It does the same thing for SQL queries: it knows that PDOStatement::execute(), PDO::query() and several other functions mean that an SQL query is being made. It groups them together, calls them sql and labels them as yellow. It's a great idea... and is just that simple.

Below this, there is another section called "Other Metrics". These are the same thing: meaningful groups of function calls. The only difference is that Blackfire does *not* give these a special color and they don't show up on the timeline. These are... just... raw data... that sit right here. If you're wondering why that would *useful*... I was too! For the purpose of the timeline, they are *not* useful. They'll come in handy later when we talk more about metrics. Metrics are their own big topic.

Finding Metrics in the Timeline

Let's look at one of the timeline metrics doctrine.entities.hydrated. What does this one mean? Sometimes the title of a metric will tell you a bit more... but often the metric name is all you really have. Most metrics are self-explanatory.

Depending on how well you know Doctrine, this might be obvious... or not. This metric refers to whenever one or more entities are *hydrated* into an object. Notice the count is 3. For this metric, it's not that there are only 3 *objects* being hydrated during this request, but that our code asks Doctrine to hydrate one or more objects on *three* occasions.

So where are the 3 times that we're hydrating objects? One of the cool things is that, when you hover over a timeline metric, it adds a border to the matching boxes on the right. It's... a little subtle... but it does the trick. I wish you could double-click and... maybe zoom to the matching boxes... but it's tricky because they may be spread out over the whole request.

If we hover over doctrine.entities.hydrated... hmm... I don't see those. You need to do a little bit of digging... I'll hover back over. There they are. It turns out that the 3 calls are *not* all in the same place: they're coming from three very different *parts* of our code. The first is part of the firewall... probably querying for the logged in user... and the other two are down in some template rendering... along with a few similarly-colored doctrine.dql.parsed items.

I want to look at what's happening inside of this template... but a lot of these things are *really* small. On top, we can see the entire timeline. Click where we want to start, move over, and let go! Zoom!

Much easier to see! In this spot, Doctrine parses its DQL, it makes an SQL query here... and a different query a bit later.

So as far as getting insight into what's *really* going on in your application, you can't get much better than this. You can even see our N+1 problem visually: it makes a query to count the comments little-by-little as the template renders.

Hit the "Home" icon to zoom back out. This is cool... but I mentioned that, as soon as we logged in, there was some *new* code that was now running in the background. Next, let's look a bit closer at the timeline to discover what that is *and* a hidden performance problem.

Chapter 16: Timeline: Finding a Hidden Surprise

One of the big spots on the timeline is the RequestEvent. It's purple because this is an event: the *first* event that Symfony dispatches. It happens before the controller is called... which is pretty obvious in this view.

Let's zoom in: by double-clicking the square. Beautiful! What happens inside this event? Apparently... the routing layer happens! That's RouterListener. You can also see Firewall: this is where authentication takes place. Security is a complex system... so being able to see a bit about what happens inside of it is pretty cool. At some point... it calls a method on EntityRepository and we can see the query for the User object that we're logged in as. Pretty cool.

The Hidden Slow Listener

There's one more big chunk under RequestEvent: something called AgreeToTermsSubscriber... which is taking 30 milliseconds. Let's open that class and see what it does: src/EventSubscriber/AgreeToTermsSubscriber.php:

73 lines src/EventSubscriber/AgreeToTermsSubscriber.php

```
... lines 1 - 2
namespace App\EventSubscriber;
use App\Entity\User;
use App\Form\AgreeToUpdatedTermsFormType;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Form\FormFactoryInterface;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\RequestEvent;
use Symfony\Component\Security\Core\Security;
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
use Twig\Environment;
class AgreeToTermsSubscriber implements EventSubscriberInterface
{
private $security;
private $formFactory;
private $twig;
private $entrypointLookup;
public function __construct(Security $security, FormFactoryInterface $formFactory, Environment $twig, EntrypointLookupInterface $entrypointLookup)
{
$this->security = $security;
$this->formFactory = $formFactory;
$this->twig = $twig;
$this->entrypointLookup = $entrypointLookup;
}
public function onRequestEvent(RequestEvent $event)
{
$user = $this->security->getUser();
// only need this for authenticated users
if (!$user instanceof User) {
return;
}
// in reality, you would hardcode the most recent "terms" date
// change so you can see if the user needs to "re-agree". I've
// set it dynamically to 1 year ago to avoid anyone hitting
// this - as it's just example code...
//$latestTermsDate = new \DateTimeImmutable('2019-10-15');
$latestTermsDate = new \DateTimeImmutable('-1 year');
$form = $this->formFactory->create(AgreeToUpdatedTermsFormType::class);
$html = $this->twig->render('main/agreeUpdatedTerms.html.twig', [
'form' => $form->createView()
]);
// resets Encore assets so they render correctly later
// only technically needed here because we should really
// "exit" this function before rendering the template if
// we know the user doesn't need to see the form!
$this->entrypointLookup->reset();
// user is up-to-date!
if ($user->getAgreedToTermsAt() >= $latestTermsDate) {
return;
}
$response = new Response($html);
$event->setResponse($response);
}
public static function getSubscribedEvents()
{
return [
RequestEvent::class => 'onRequestEvent',
];
}
}
```

Ah yes. Every now and then, we update the "terms of service" on our site. When we do that, our lovely lawyers have told us that we need to require people

to agree to the updated terms. *This* class handles that: it gets the authenticated user and, if they're not logged in, it does nothing:

73 lines src/EventSubscriber/AgreeToTermsSubscriber.php

```
... lines 1 - 14
class AgreeToTermsSubscriber implements EventSubscriberInterface
{
... lines 17 - 29
public function onRequestEvent(RequestEvent $event)
{
$user = $this->security->getUser();
// only need this for authenticated users
if (!$user instanceof User) {
return;
}
... lines 38 - 63
}
... lines 65 - 71
}
```

But if they *are* logged in, then it renders a twig template with an "agree to the terms" form:

73 lines src/EventSubscriber/AgreeToTermsSubscriber.php

```
... lines 1 - 14
class AgreeToTermsSubscriber implements EventSubscriberInterface
{
... lines 17 - 29
public function onRequestEvent(RequestEvent $event)
{
... lines 32 - 38
// in reality, you would hardcode the most recent "terms" date
// change so you can see if the user needs to "re-agree". I've
// set it dynamically to 1 year ago to avoid anyone hitting
// this - as it's just example code...
//$latestTermsDate = new \DateTimeImmutable('2019-10-15');
$latestTermsDate = new \DateTimeImmutable('-1 year');
$form = $this->formFactory->create(AgreeToUpdatedTermsFormType::class);
$html = $this->twig->render('main/agreeUpdatedTerms.html.twig', [
'form' => $form->createView()
]);
// resets Encore assets so they render correctly later
// only technically needed here because we should really
// "exit" this function before rendering the template if
// we know the user doesn't need to see the form!
$this->entrypointLookup->reset();
// user is up-to-date!
if ($user->getAgreedToTermsAt() >= $latestTermsDate) {
return;
}
... lines 61 - 63
}
... lines 65 - 71
}
```

Eventually, *if* the terms have been updated since the last time this User agreed to them, it sets that form as the response instead of rendering the *real* page.

We haven't seen this form yet... and... it's not really that important. Because we *rarely* update our terms, 99.99% of the requests to the site will *not* display the form.

So... the fact that this is taking 30 milliseconds... even though it will almost *never* do anything... is kind of a lot!

Blue Memory Footprint

Oh, and see this blue background? I love this: it's the memory footprint. If we trace over this call - this is about when the AgreeToTermsSubscriber happens - the memory starts at 3.44 megabytes... and finishes around 4.46. That's 1 megabyte of memory - kinda high for such a rarely-used function.

The point is: this method doesn't take *that* long to run. And so, it may not have shown up as a performance critical path on the call graph. But thanks to the timeline, this invisible layer jumped out at us. And... I think it *is* taking a bit too long.

Fixing the Slow Code

Back in the code, the mistake I made is pretty embarrassing. I'm using some pretend logic to see whether or not we need to render the form. But... I put the check too late!

73 lines src/EventSubscriber/AgreeToTermsSubscriber.php

```
... lines 1 - 14
class AgreeToTermsSubscriber implements EventSubscriberInterface
{
... lines 17 - 29
public function onRequestEvent(RequestEvent $event)
{
... lines 32 - 56
// user is up-to-date!
if ($user->getAgreedToTermsAt() >= $latestTermsDate) {
return;
}
... lines 61 - 63
}
... lines 65 - 71
}
```

We're doing all the work of rendering the form... even if we don't use it.

Let's move that code all the way to the top. Ah, too far - it needs to be after the fake $latestTermsDate variable:

73 lines src/EventSubscriber/AgreeToTermsSubscriber.php

```
... lines 1 - 14
class AgreeToTermsSubscriber implements EventSubscriberInterface
{
... lines 17 - 29
public function onRequestEvent(RequestEvent $event)
{
... lines 32 - 38
// in reality, you would hardcode the most recent "terms" date
// change so you can see if the user needs to "re-agree". I've
// set it dynamically to 1 year ago to avoid anyone hitting
// this - as it's just example code...
//$latestTermsDate = new \DateTimeImmutable('2019-10-15');
$latestTermsDate = new \DateTimeImmutable('-1 year');
// user is up-to-date!
if ($user->getAgreedToTermsAt() >= $latestTermsDate) {
return;
}
... lines 50 - 63
}
... lines 65 - 71
}
```

That looks better. Let's try it! I'll refresh the page. Profile again and call it [Recording] Homepage authenticated fixed subscriber: http://bit.ly/sf-bf-timeline-fix

Let's jump straight to view the Timeline... double-click RequestEvent and this time... AgreeToTermsSubscriber is gone! We can see RouterListener and Firewall... but *not* AgreeToTermsSubscriber. That's not because our app isn't *calling* it anymore: it is. It's because Blackfire hides function calls that take almost no resources. That's great.

Next, we know that we can write code inside a function that is slow. But did you know that sometimes even the *instantiation* of an object can eat a lot of resources? Let's see how that looks in Blackfire and leverage a Symfony feature - service subscribers - to make instantiation lighter.

Chapter 17: Spotting Heavy Object Instantiation

I want to show a... more *subtle* performance problem. To even *see* it, we need to go back to the prod environment:

29 lines .env

```
... lines 1 - 16
APP_ENV=prod
... lines 18 - 29
```

Make sure to run cache:clear:

php bin/console cache:clear

cache:warmup:

php bin/console cache:warmup

And also:

composer dump-autoload --optimize

Let's create a fresh profile of the homepage. I'll call this one: [Recording] Homepage prod. Click to view the timeline: http://bit.ly/sf-bf-instantiation

Overall, this request is pretty fast. Click into the "Memory" dimension. The biggest call is Composer\Autoload\includeFile: that's *literally* Composer including files that we need... not a lot of memory optimization we can do about that.

But, if we look closer, the memory dimension reveals something else. See this "Container" thing - the 2nd item on the function list? This is related to Symfony's container, which is responsible for *instantiating* all of our objects. This specific function is interesting: it's highlighting a *section* of a file that lives in our cache directory. If you looked in that file, you would see that this part of the code is responsible for *including* some of the main files that our app needs. It's basically another version of the top node: it's code that includes files for classes we're using.

Seeing Object Instantiation

Ok, so the first few aren't really *that* interesting. Things get much more intriguing down on the 4th function call: some Container{BlahBlah}/getDoctrine_Orm_DefaultEntityManagerService.php call. What is this? Well, the details of how this is organized are specific to Symfony: but this is evidence of something that *every* app does: this is showing the amount of resources used to *instantiate* Doctrine's EntityManager object. I know, we don't often think about how much time or how much memory it takes to *instantiate* an object, but it *can* sometimes be a problem. The next function call is for the instantiation of Doctrine's Connection service.

Go down a little bit... I'm looking for something specific... here it is: getLoginFormAuthenticatorService(). This is responsible for instantiating a LoginFormAuthenticator object in our app. It's not a particularly problematic function though: it's 10th on the list... only takes 2.56 milliseconds and uses about 500 kilobytes.

Checking the Instantiation of LoginFormAuthenticator

Let's check out the class: src/Security/LoginFormAuthenticator.php:

96 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 2
namespace App\Security;
... lines 4 - 21
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
use TargetPathTrait;
private $entityManager;
private $urlGenerator;
private $csrfTokenManager;
private $passwordEncoder;
public function __construct(EntityManagerInterface $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface
$csrfTokenManager, UserPasswordEncoderInterface $passwordEncoder)
{
$this->entityManager = $entityManager;
$this->urlGenerator = $urlGenerator;
$this->csrfTokenManager = $csrfTokenManager;
$this->passwordEncoder = $passwordEncoder;
}
public function supports(Request $request)
{
```

```
return 'app_login' === $request->attributes->get('_route')
&& $request->isMethod('POST');
}
public function getCredentials(Request $request)
{
$credentials = [
'email' => $request->request->get('email'),
'password' => $request->request->get('password'),
'csrf_token' => $request->request->get('_csrf_token'),
];
$request->getSession()->set(
Security::LAST_USERNAME,
$credentials['email']
);
return $credentials;
}
public function getUser($credentials, UserProviderInterface $userProvider)
{
$token = new CsrfToken('authenticate', $credentials['csrf_token']);
if (!$this->csrfTokenManager->isTokenValid($token)) {
throw new InvalidCsrfTokenException();
}
$user = $this->entityManager->getRepository(User::class)->findOneBy(['email' => $credentials['email']]);
if (!$user) {
// fail authentication with a custom error
throw new CustomUserMessageAuthenticationException('Email could not be found.');
}
return $user;
}
public function checkCredentials($credentials, UserInterface $user)
{
return $this->passwordEncoder->isPasswordValid($user, $credentials['password']);
}
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
return new RedirectResponse($targetPath);
}
return new RedirectResponse($this->urlGenerator->generate('app_homepage'));
}
protected function getLoginUrl()
{
return $this->urlGenerator->generate('app_login');
}
}
```

As its name suggests, this is responsible for authenticating the user when they submit the login form.

But, there's something special about this class. Due to the way the Symfony security system works, Symfony instantiates this object on *every* request. It does that so it can then call supports() to figure out if this service should be "activated" on this request or not:

96 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 21
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 24 - 38
public function supports(Request $request)
{
return 'app_login' === $request->attributes->get('_route')
&& $request->isMethod('POST');
}
... lines 44 - 94
}
```

For this class, it *only* needs to its work when the URL is /login and this is a POST request. In *every* other situation, supports() returns false and *no* other methods are called on this class.

So let's think about this. Instantiating this class takes about 3 milliseconds and 500 kilobytes... which is not a *ton*... but since *all* it needs to do for *most* requests is check the current URL... then exit... that *is* kind of heavy.

Why Instantiation is Slow?

The question is: *why* does it take so many resources to instantiate? Well, 500 kilobytes is not a *ton*, but this *is* - according to Blackfire - one of the *most* expensive objects that is created on this request. Why?

Check out the constructor:

96 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 21
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 24 - 30
public function __construct(EntityManagerInterface $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface
$csrfTokenManager, UserPasswordEncoderInterface $passwordEncoder)
{
$this->entityManager = $entityManager;
$this->urlGenerator = $urlGenerator;
$this->csrfTokenManager = $csrfTokenManager;
$this->passwordEncoder = $passwordEncoder;
}
... lines 38 - 94
}
```

In order to instantiate this class, Symfony needs to make sure the EntityManager is instantiated... and the UrlGenerator.. and the CsrfTokenManager... and the UserPasswordEncoder. If any of *these* services have their *own* dependencies, even *more* objects may need to be instantiated. In rare situations, creating a service can be a *huge* performance problem.

In the case of the EntityManager and the UrlGenerator... those are pretty core objects that would *probably* be needed and thus instantiated by *something* on this request anyways. But CsrfTokenManager and UserPasswordEncoder are *not* normally needed. In other words, we're forcing Symfony to instantiate both of those services on *every* request... even though we *only* need them when the user is submitting the login form.

This is a *classic* situation where you have an object that is instantiated on every request... but only needs to do *real* work in rare cases. Certain event subscribers - like our AgreeToTermsSubscriber - Symfony security voters & Twig extensions are other examples from Symfony. These services might be quick to instantiate... so no problem! But they *also* might be expensive.

So... how *could* we make it quicker to instantiate LoginFormAuthenticator? In Symfony, with a service subscriber.

Chapter 18: Service Subscribers

Because this service is instantiated on every request... it means that all four of the objects in its constructor *also* need to be instantiated:

96 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 21
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 24 - 30
public function __construct(EntityManagerInterface $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface
$csrfTokenManager, UserPasswordEncoderInterface $passwordEncoder)
{
$this->entityManager = $entityManager;
$this->urlGenerator = $urlGenerator;
$this->csrfTokenManager = $csrfTokenManager;
$this->passwordEncoder = $passwordEncoder;
}
... lines 38 - 94
}
```

That's not a *huge* deal... except that two of these services *probably* wouldn't be instantiated during a normal request *and* aren't even used unless the current request is a login form submit. In other words, we're *always* instantiating these objects... even though we don't need them!

How can we fix this? By using a service subscriber: it's a strategy in Symfony that allows you to get a service you need... but *delay* its instantiation until - and unless - you actually need to use it. It's great for performance. But, like many things, it comes at a cost: a bit more complexity.

Implementing ServiceSubscriberInterface

Start by adding an interface to this class: ServiceSubscriberInterface:

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 21
use Symfony\Contracts\Service\ServiceSubscriberInterface;
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 100
}
```

Then I'll move to the bottom of the file, go to the "Code"->"Generate" menu - or Command+N on a Mac - and select "Implement Methods" to generate the *one* method this interface requires: getSubscribedServices():

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 23
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 91
public static function getSubscribedServices()
{
... lines 94 - 99
}
}
```

What does this return? An array of type-hints for all the services we need. For this class, it's these four. So, return EntityManagerInterface::class, UrlGeneratorInterface::class, CsrfTokenManagerInterface::class and OtherLongInterfaceName::class, uh, UserPasswordEncoderInterface::class:

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 23
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 91
public static function getSubscribedServices()
{
return [
EntityManagerInterface::class,
UrlGeneratorInterface::class,
CsrfTokenManagerInterface::class,
UserPasswordEncoderInterface::class,
];
}
}
```

By doing this, we can now *remove* these four arguments. Replace them with ContainerInterface - the one from Psr\Container - $container:

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 6
use Psr\Container\ContainerInterface;
... lines 8 - 23
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 29
public function __construct(ContainerInterface $container)
{
... line 32
}
... lines 34 - 100
}
```

When Symfony sees the new interface and this argument, it will pass us a, sort of, "mini-container" that holds the 4 objects we need. But it does this in a way where those 4 objects aren't *created* until we use them.

Finish this by removing the old properties... and having just one: $container. Set it with $this->container = $container:

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 23
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 27
private $container;
public function __construct(ContainerInterface $container)
{
$this->container = $container;
}
... lines 34 - 100
}
```

Using the Container Locator

Because those properties are gone, *using* the services looks a bit different. For example, down here for CsrfTokenManager, now we need to say $this->container->get() and pass it the type-hint CsrfTokenManagerInterface::class:

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 23
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 55
public function getUser($credentials, UserProviderInterface $userProvider)
{
... line 58
if (!$this->container->get(CsrfTokenManagerInterface::class)->isTokenValid($token)) {
... line 60
}
... lines 62 - 70
}
... lines 72 - 100
}
```

This will work *just* like before *except* that the CsrfTokenManager won't be instantiated *until* this line is hit... and if this line *isn't* hit, it *won't* be instantiated.

For entityManager, use $this->container->get(EntityManagerInterface::class), for passwordEncoder, $this->container->get(UserPasswordEncoderInterface::class) and finally, for urlGenerator, use $this->container->get->(UrlGeneratorInterface::class). I'll copy that and use it again inside getLoginUrl():

102 lines src/Security/LoginFormAuthenticator.php

```
... lines 1 - 23
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements ServiceSubscriberInterface
{
... lines 26 - 55
public function getUser($credentials, UserProviderInterface $userProvider)
{
... line 58
if (!$this->container->get(CsrfTokenManagerInterface::class)->isTokenValid($token)) {
... line 60
}
$user = $this->container->get(EntityManagerInterface::class)->getRepository(User::class)->findOneBy(['email' => $credentials['email']]);
... lines 64 - 70
}
public function checkCredentials($credentials, UserInterface $user)
{
return $this->container->get(UserPasswordEncoderInterface::class)->isPasswordValid($user, $credentials['password']);
}
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
... lines 80 - 83
return new RedirectResponse($this->container->get(UrlGeneratorInterface::class)->generate('app_homepage'));
}
protected function getLoginUrl()
{
return $this->container->get(UrlGeneratorInterface::class)->generate('app_login');
}
... lines 91 - 100
}
```

So, a *little* bit more complicated... but it *should* take less resources to create this class. The question is: did this make *enough* difference for us to *want* this added complexity? Let's find out. First, clear the cache:

php bin/console cache:clear

And warm it up:

php bin/console cache:warmup

Comparing the Results

Move back over... I'll close some tabs and... refresh. Profile again: I'll call this one: [Recording] Homepage service subscriber: https://bit.ly/sf-bf-service-subscriber. View the call graph.

Excellent! Go back to the "Memory" dimension and search for "login". The call is still here but it's taking a lot less memory *and* less time. Let's compare this to be sure though. Click back to the homepage and go from the previous profile to this one: https://bit.ly/sf-bf-service-subscriber-compare.

Nice! The wall time is down by 4%... CPU is down and memory *also* decreased... but *just* a little bit.

So was this change worth it? Probably. But this doesn't mean you should run around and use service subscribers *everywhere*. Why? Because they add complexity to your code *and*, unless you have a specific situation, it won't help much or at all. Use Blackfire to find the *real* problems and target those.

For example, we also could have made this same change to our AgreeToTermsSubscriber:

73 lines src/EventSubscriber/AgreeToTermsSubscriber.php

```
... lines 1 - 2
namespace App\EventSubscriber;
... lines 4 - 14
class AgreeToTermsSubscriber implements EventSubscriberInterface
{
private $security;
private $formFactory;
private $twig;
private $entrypointLookup;
public function __construct(Security $security, FormFactoryInterface $formFactory, Environment $twig, EntrypointLookupInterface $entrypointLookup)
{
$this->security = $security;
$this->formFactory = $formFactory;
$this->twig = $twig;
$this->entrypointLookup = $entrypointLookup;
}
public function onRequestEvent(RequestEvent $event)
{
$user = $this->security->getUser();
// only need this for authenticated users
if (!$user instanceof User) {
return;
}
// in reality, you would hardcode the most recent "terms" date
// change so you can see if the user needs to "re-agree". I've
// set it dynamically to 1 year ago to avoid anyone hitting
// this - as it's just example code...
//$latestTermsDate = new \DateTimeImmutable('2019-10-15');
$latestTermsDate = new \DateTimeImmutable('-1 year');
// user is up-to-date!
if ($user->getAgreedToTermsAt() >= $latestTermsDate) {
return;
}
$form = $this->formFactory->create(AgreeToUpdatedTermsFormType::class);
$html = $this->twig->render('main/agreeUpdatedTerms.html.twig', [
'form' => $form->createView()
]);
// resets Encore assets so they render correctly later
// only technically needed here because we should really
// "exit" this function before rendering the template if
// we know the user doesn't need to see the form!
$this->entrypointLookup->reset();
$response = new Response($html);
$event->setResponse($response);
}
public static function getSubscribedEvents()
{
return [
RequestEvent::class => 'onRequestEvent',
];
}
}
```

This class is *also* instantiated on every request... but rarely needs to do its work. That means we are causing the FormFactory object to be instantiated on every request.

But, go back to the latest profile... click to view the memory dimension... and search for "agree". There it is! It took 1.61 milliseconds and 41 kilobytes to

instantiate this. That's... a lot less than the login authenticator. So, is making this class a service subscriber worth it? For me, no. I'd rather get back to writing features or fixing bigger performance issues.

Next, we can take a lot more control of the profiling process, like profiling just a *portion* of our code *or* automatically triggering a profile based on some condition, instead of needing to manually use the browser extension. Let's talk about the Blackfire SDK next.

Chapter 19: Manually Profile (Instrument) Part of your Code

Profiling a page looks like this.

Profiling: What happens Behind the Scenes

First, something tells the Blackfire PHP extension - the "Probe":

> Hey! Start profiling!

Which basically means that it starts collecting *tons* of data. The process of collecting data is called *instrumentation*... because when a concept is *too* simple, sometimes we tech people like to invent confusing words. *Instrumentation* means that the PHP extension is collecting data.

The second step is that - eventually - something tells the PHP extension to *stop* "instrumentation" and to send the data. The collection of data is known as a "profile". The PHP extension sends the profile to the agent, which aggregates it, prune some stuff and ultimately sends it to the Blackfire server.

So: what is the "thing" that tells the PHP extension to activate? We know that the PHP extension doesn't profile *every* request... so what is it that says:

> Hey PHP extension "probe" thing: start profiling!

The answer - *so* far - is: the browser extension: it sends special information that tells the probe to do its thing. Or, if you use the blackfire command line utility, which we did earlier to profile a command, then *it* is what tells the PHP extension to activate.

In either situation, the extension is activated *before* even the *first* line of code is executed. That means that *every* single line of PHP code is "instrumented": our final profile contains *everything*. This is called auto-instrumentation: instrumentation starts automatically.

This naturally leads to three interesting questions.

First, who *is* baby Yoda? I mean, is he... like, related to Yoda? Or just the same species?

The second question is: could we trigger, or *create* a Blackfire profile in a *different* way? Could we, for example, dynamically tell the PHP extension to create a profile from *inside* our code under some specific condition?

And third, *regardless* of who *triggers* the profile, could we "zoom in" and only collect profiling data for *part* of our code? Like, could we create a profile that only collects data about the code from our controller instead of the entire request?

Let's actually start with that last question: profiling a *specific* part of our code, instead of the whole thing. To be *fully* honest, I don't know if this part has a *ton* of practical use-cases, but it *will* give you an even better idea of how Blackfire works behind the scenes.

Installing the Blackfire SDK

To help with this crazy experiment, we're going to install Blackfire's PHP SDK. Find your terminal, dial up your modem to the Internet, and run:

```
composer require blackfire/php-sdk
```

This is a normal PHP library that helps interact directly with Blackfire from *inside* your code. You'll see how.

When it finishes, move over and open src/Controller/MainController.php:

122 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
/**
* @Route("/", name="app_homepage")
*/
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
$sightings = $this->createSightingsPaginator(1, $bigFootSightingRepository);
return $this->render('main/homepage.html.twig', [
'sightings' => $sightings
]);
}
... lines 30 - 120
}
```

Ok: this is the controller for our homepage. Let's pretend that when we profile this page, we don't want to collect data about *all* of our code. Nope, we want to, sort of, "zoom in" and see *only* what's happening inside the controller.

Manually Instrumenting Code

We can do that by saying $probe = \BlackfireProbe::getMainInstance():

128 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 21
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
$probe = \BlackfireProbe::getMainInstance();
... lines 25 - 34
}
... lines 36 - 126
}
```

Remember: the PHP extension is called the "probe"... that's important if you want this to make sense. Then call $probe->enable():

128 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 21
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
$probe = \BlackfireProbe::getMainInstance();
$probe->enable();
... lines 26 - 34
}
... lines 36 - 38
*/
... lines 40 - 126
}
```

At the bottom, I'll set the rendered template to a $response variable, add $probe->disable() and finish with return $response:

128 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 21
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
$probe = \BlackfireProbe::getMainInstance();
$probe->enable();
... lines 26 - 27
$response = $this->render('main/homepage.html.twig', [
'sightings' => $sightings
]);
$probe->disable();
return $response;
}
... lines 36 - 126
}
```

Okay, so... what the heck does this do? The first thing I want you to notice is that if I refresh the homepage a bunch of times... and then go to https://blackfire.io, I do *not* have any new profiles. Adding this code does not "trigger" a new profile to be created: it does *not* tell the PHP extension - the "probe" - that it should to do its work.

Instead, *if* a profile is currently being created, this tells the probe *when* to start collecting data. Hmm, this isn't going to *quite* make sense until we see it in action. Trigger a new profile on the homepage. I'll call this one: [Recording] Only instrument some code.

Click to view the call graph: https://bit.ly/sf-bf-partial-profile.

Fascinating. This contains *less* information than normal. It has a few things on top - main() and handleRaw()... but basically it jumps straight to the homepage() method.

How Disabling Auto-Instrumentation Works

What's happening here is that the *only* code that the probe "instrumented", the *only* code that it collected information on, is the code between the enable() and disable() calls:

128 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 21
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
... line 24
$probe->enable();
$sightings = $this->createSightingsPaginator(1, $bigFootSightingRepository);
$response = $this->render('main/homepage.html.twig', [
'sightings' => $sightings
]);
$probe->disable();
... lines 33 - 34
}
... lines 36 - 126
}
```

This... completely confused me the first time I saw it. What *really* happens is this: as soon as we use the browser extension to tell the probe to do its job, the PHP extension starts instrumenting - so, collection data - *immediately*. Initially, it *is* collecting data about *every* line of PHP code.

But as *soon* as it sees $probe->enable(), it basically *forgets* about all the data collected so far. The $probe->enable() call says:

> Hey! Start instrumenting *here*. If you've already collected some data before thanks to auto-instrumentation, get rid of it.

This effectively *disables* auto-instrumentation: we're now *controlling* which code is instrumented instead of it happening automatically. Once the code hits $probe->disable() instrumentation stops.

You can actually use $probe->enable() and $probe->disable() multiple times in your code if you want to profile different pieces: $probe->enable() only forgets data it's already collected the *first* time you call it.

Oh, and you can *also* optionally call $probe->close() - you'll see this in their documentation:

129 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 21
public function homepage(BigFootSightingRepository $bigFootSightingRepository)
{
... lines 24 - 31
$probe->disable();
$probe->close(); // optional - will auto-close at end of script
... lines 34 - 35
}
... lines 37 - 127
}
```

That tells the PHP extension that you're *definitely* done profiling and it can send the data to the agent. But, it's not *strictly* required, because it'll be sent automatically when the script ends anyways.

So... this feature is *maybe* useful... but it's *definitely* a nice intro into taking more control of the profiling process.

We haven't used the SDK Yet

And.. fun fact! We installed the blackfire/php-sdk library... but we haven't *actually* used it yet! This \BlackfireProbe class is *not* from the php-sdk library: it's from the Blackfire PHP *extension*. As long as you have the extension installed, that class will exist. We're interacting *directly* with the extension.

So... why did we install the SDK if we didn't need it? Because... it gave us auto-complete on that class. And you all know that I freakin' *love* auto-complete.

The SDK has a, sort of, "stub" version of this class. This is *not* the code that was *actually* executed when we called those methods... but having this at least shows us what methods and arguments exist.

Next, let's actually *use* the PHP SDK to do something a bit more interesting. I want to *create* a profile automatically in my code *without* needing to use the browser extension. This *does* have real-world use-cases.

Chapter 20: SDK: Automatically Create a Profile

Imagine you have a performance "problem" on production. No worries! Except... the issue is only caused in some edge-case situation... and you're having a hard time repeating the *exact* condition... which means that you can't create a meaningful Blackfire profile by using the browser extension.

For example, imagine we want to profile the AJAX request that loads the GitHub repository info... but we think that the performance problem only happens for certain *types* of users - maybe users that have *many* comments. I'm just making this up.

To do that, *instead* of triggering a new profile by clicking the browser extension button - which maybe is hard because we can't seem to replicate the correct situation - let's trigger a new profile automatically from *inside* our code. We can do this using the PHP SDK.

Spin over, go back to MainController and scroll down to loadSightingsPartial()... actually to the gitHubOrganizationInfo() method:

129 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 58
/**
* @Route("/api/github-organization", name="app_github_organization_info")
*/
public function gitHubOrganizationInfo(GitHubApiHelper $apiHelper)
{
$organizationName = 'SymfonyCasts';
$organization = $apiHelper->getOrganizationInfo($organizationName);
$repositories = $apiHelper->getOrganizationRepositories($organizationName);
return $this->json([
'organization' => $organization,
'repositories' => $repositories,
]);
}
... lines 73 - 127
}
```

This is the controller that returns the content on the right side of the page.

Start by creating a fake variable $shouldProfile = true:

131 lines src/Controller/MainController.php

```
... lines 1 - 17
class MainController extends AbstractController
{
... lines 20 - 55
public function gitHubOrganizationInfo(GitHubApiHelper $apiHelper)
{
// replace with some conditional logic
$shouldProfile = true;
... lines 60 - 73
}
... lines 75 - 129
}
```

In a real app, you would replace this with logic to determine whether or not this is one of those requests that you think might have a performance problem: maybe you check to see if the user has *many* comments or something.

Creating & Starting the Profile

Then, if $shouldProfile, it means that we *want* Blackfire to profile this request. To do that, say $blackfire = new Client() - the one from Blackfire. This is an object that helps communicate with the Blackfire servers. Next, *create* a probe - basically create a new "profile" - with $probe = $blackfire->createProbe():

131 lines src/Controller/MainController.php

```
... lines 1 - 17
class MainController extends AbstractController
{
... lines 20 - 55
public function gitHubOrganizationInfo(GitHubApiHelper $apiHelper)
{
// replace with some conditional logic
$shouldProfile = true;
if ($shouldProfile) {
$blackfire = new Client();
$probe = $blackfire->createProbe();
}
... lines 65 - 73
}
... lines 75 - 129
}
```

Earlier, when we used BlackfireProbe::getMainInstance(), we were, kind of *asking* for a "probe" if there *was* a profile happening. But this time, we're *creating* a probe: creating a new profile and telling it to start "instrumenting" - collecting data - right now.

In fact, the second argument to createProbe() is $enabled=true: whether or not we want the probe to *immediately* start instrumentation or if we will enable it later with $probe->enable().

Now, *because* this will *create* a new profile, you need to make sure you do this only *rarely* on production. Why? Because creating profiles is heavy and this slow request will be *felt* by whichever user triggered it. So, choose your logic for $shouldProfile carefully.

Anyways, let's try it! Move over and refresh your list of Blackfire profiles. The most recent one is the "Only instrumenting some code" profile. Now refresh the homepage. This triggers the AJAX call... but notice it's slower. And when we refresh Blackfire... boom! We have a brand new profile! Open that up and... let's give it a name: [Recording] First automatic profile: http://bit.ly/f-bf-1st-auto-profile. I'm so proud.

This only Profiles the Controller

You can now create *new* profiles from your code... *whenever* you want to. But... there's a small problem: this only profiled a *tiny* part of our code. And, that makes sense: when our PHP code started executing, the PHP extension didn't yet know that we wanted to profile this request. And so, it couldn't start collecting data until we told it to, which happened in the controller. To make matters *worse*, as *soon* as PHP garbage collected the $probe variable... which happened once the variable isn't used anymore... so at the end of the controller, internally, the probe called close() on itself. That means that we just collected data on *nothing* more than the code in our controller.

How can we fix that? By starting the probe *super* early and closing it manually as late as we can. Let's do that next.

Chapter 21: Creating an Automatic Probe Early in your Code

Once we determine that we want to *create* a probe dynamically in our code, we *really* want to do that as *early* as possible so that Blackfire can "instrument" as much of our *code* as possible.

Generating the Event Subscriber

In Symfony, we can do that with an event subscriber... which we will *generate* to be super lazy. First, in .env, make sure that you're back in the dev environment:

29 lines .env

```
... lines 1 - 16
APP_ENV=dev
... lines 18 - 29
```

Then, find your terminal and run:

php bin/console make:subscriber

Call it BlackfireAutoProfileSubscriber... and we want to listen to RequestEvent: Go check out the code src/EventSubscriber/BlackfireAutoProfileSubscriber.php:

22 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 2
namespace App\EventSubscriber;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\RequestEvent;
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
public function onRequestEvent(RequestEvent $event)
{
// ...
}
public static function getSubscribedEvents()
{
return [
RequestEvent::class => 'onRequestEvent',
];
}
}
```

So, when this RequestEvent happens - which Symfony dispatches *super* early when handling a request, we want to create & enable the probe. Copy all of the $shouldProfile code, remove it from the controller and paste it here:

30 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 4
use Blackfire\Client;
... lines 6 - 8
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
public function onRequestEvent(RequestEvent $event)
{
... lines 13 - 16
if ($shouldProfile) {
$blackfire = new Client();
$probe = $blackfire->createProbe();
}
}
... lines 22 - 28
}
```

Creating the Prove in the Subscriber

Now add $request = $event->getRequest(). To make this *only* profile the GitHub organization AJAX call - whose URL is /api/github-organization - set $shouldProfile equal to $request->getPathInfo() === '/api/github-organization':

30 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 8
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
public function onRequestEvent(RequestEvent $event)
{
// replace with some conditional logic
$request = $event->getRequest();
$shouldProfile = $request->getPathInfo() === '/api/github-organization';
if ($shouldProfile) {
$blackfire = new Client();
$probe = $blackfire->createProbe();
}
}
... lines 22 - 28
}
```

In a real app, I would add *more* code to make sure $shouldProfile is *only* true on the *very* specific requests we want to profile.

Now I'll re-type the t on Client and select the correct Client class so that PhpStorm adds that use statement to the top of the class for me:

30 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 4
use Blackfire\Client;
... lines 6 - 8
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
public function onRequestEvent(RequestEvent $event)
{
... lines 13 - 16
if ($shouldProfile) {
$blackfire = new Client();
... line 19
}
}
... lines 22 - 28
}
```

Thanks PhpStorm!

But before we try this, I want to code for one edge case: if *not* $event->isMasterRequest(), then return:

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 10
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 13 - 17
public function onRequestEvent(RequestEvent $event)
{
if (!$event->isMasterRequest()) {
return;
}
... lines 23 - 31
}
... lines 33 - 48
}
```

It might not be important in your app, but Symfony has a "sub-request" system... and the short explanation is that we don't want to profile those: they are not *real* requests... and would make a big mess of things.

Ok, let's try this! I'll close a tab... then refresh the homepage... which causes the AJAX request to be made. You can see it's slow. Now reload the list of profiles on Blackfire... there it is! Open it up.

And... oh wow, oh weird! 281 *microseconds*. Give this a name: [Recording] Auto from subscriber: http://bit.ly/sf-bf-broken-auto-profile

This profile is... broken. That's 281 *microseconds* - so .281 milliseconds. And the entire profile is just the Probe::enable() call itself!

Probe Auto-Close Too Early

What happened!? Well... remember: the $probe object automatically calls close() on *itself* as soon as that variable is garbage collected... which happens at the end of the subscriber method. That means.... we profiled exactly *one* line of code.

The solution is to call $probe->close() manually... which - more importantly - will require us to store the Probe object in a way where PHP *won't* garbage collect it too early.

So here's the goal: call $probe->close() as *late* as possible during the request lifecycle. We can do this by listening to a *different* event: when TerminateEvent::class is dispatched - that's *very* late in Symfony - call the onTerminateEvent() method:

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 8
use Symfony\Component\HttpKernel\Event\TerminateEvent;
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 13 - 40
public static function getSubscribedEvents()
{
return [
... lines 44 - 45
TerminateEvent::class => 'onTerminateEvent',
];
}
}
```

I'll hit an Alt+Enter shortcut to create that method... then add the argument TerminateEvent $event:

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 10
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 13 - 33
public function onTerminateEvent(TerminateEvent $event)
{
... lines 36 - 38
}
... lines 40 - 48
}
```

To be able to call $probe->close(), we need to store the probe object on a property. Add private $probe with some documentation that says that this will either be a Probe instance from Blackfire or null:

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 10
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
/**
* @var Probe|null
*/
private $probe;
... lines 17 - 48
}
```

Update the code below to be $this->probe = $blackfire->createProbe():

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 10
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
/**
* @var Probe|null
*/
private $probe;
... line 17
public function onRequestEvent(RequestEvent $event)
{
... lines 20 - 27
if ($shouldProfile) {
... line 29
$this->probe = $blackfire->createProbe();
}
}
... lines 33 - 48
}
```

Finally, inside onTerminateEvent, if $this->probe - I should *not* have put that exclamation point, that's a mistake - then $this->probe->close():

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 10
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 13 - 33
public function onTerminateEvent(TerminateEvent $event)
{
if ($this->probe) {
$this->probe->close();
}
}
... lines 40 - 48
}
```

If you assume that I did *not* include the exclamation point... then this makes sense! *If* we created the probe, then we will close it. Problem solved. And... *really*... the fact that we set the probe onto a *property* is the real magic: that will prevent PHP from garbage-collecting that object... which will prevent it from closing itself until we're ready.

Increasing the Event Priority

While we're here, let's make this a little bit cooler. Change onRequestEvent to be an array... and add 1000 as the second item:

50 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 10
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 13 - 40
public static function getSubscribedEvents()
{
return [
// warning: adding a priority will run before routing & security
RequestEvent::class => ['onRequestEvent', 1000],
... line 46
];
}
}
```

This syntax is... weird. But the result is good: it says that we want to listen to this event with a priority of 1000. That will make our code run even *earlier* so that even *more* code will get profiled.

Configuration: Name your Profile

Oh, and there's one other cool thing we can do: we can *configure* the profile. Add $configuration = new Configuration() from Blackfire\Profile. Thanks to this, we can control a number of things about the profile... the best being ->setTitle(): Automatic GitHub org Profile. Pass this to createProbe():

53 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 6
use Blackfire\Profile\Configuration;
... lines 8 - 11
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 14 - 18
public function onRequestEvent(RequestEvent $event)
{
... lines 21 - 28
if ($shouldProfile) {
$configuration = new Configuration();
$configuration->setTitle('Automatic GitHub org profile');
$blackfire = new Client();
$this->probe = $blackfire->createProbe($configuration);
}
}
... lines 36 - 51
}
```

That's it! Let's see how things whole thing works. Back at the browser, I'll close the old profile... and refresh the homepage. Once the AJAX call finishes... reload the Blackfire profile list. Ah! We were too fast - it's still processing. Try again and... there it is!

Open it up! http://bit.ly/sf-bf-auto-profile-subscriber

Much better. A few things might still look a *bit* odd... because we're still not profiling *every* single line of code. For example, Probe::enable() seems to wrap everything. But all the important data is there.

To avoid making a *million* of these profiles as we keep coding, I'll go back to the subscriber and avoid profiling entirely by setting $shouldProfile = false:

56 lines src/EventSubscriber/BlackfireAutoProfileSubscriber.php

```
... lines 1 - 11
class BlackfireAutoProfileSubscriber implements EventSubscriberInterface
{
... lines 14 - 18
public function onRequestEvent(RequestEvent $event)
{
... lines 21 - 28
// stop our testing code from profiling
$shouldProfile = false;
if ($shouldProfile) {
... lines 33 - 36
}
}
... lines 39 - 54
}
```

Next: you already write automated tests for your app to help *prove* that key features never have bugs. You... ah... do write tests right? Let's... say you do. Me too.

Anyways, have you ever thought about writing automated tests to prevent *performance* bugs? Yep, that's possible! We can use Blackfire *inside* our test suite to add performance *assertions*. It's pretty sweet... and now that we understand the SDK, it will feel great.

Chapter 22: Performance Tests

Let's profile the Github API endpoint again. I'll cheat and go directly to /api/github-organization... and click to profile this. I'll call it: [Recording] GitHub Ajax HTTP requests because we're going to look closer at the HTTP requests that our app makes to the GitHub API.

Click to view the call graph: https://bit.ly/sf-bf-http-requests

Oh wow - this request was *super* slow - 1.83 seconds - a lot slower than we've seen before. We can see that curl_multi_select() is the problem: this is our code making requests to the GitHub API, which is *apparently* running a bit slow at the moment.

We have a Performance "Bug"

Lucky for us, that's *exactly* what I wanted to talk about! At the top, Blackfire tells me that this page made *two* HTTP requests. And HTTP requests are *always* expensive for performance.

If you studied the data from the two API endpoints that we're using, you would discover that it's *possible* - by writing some clever code - to get *all* the info our app needs with just *one* HTTP request.

What I'm saying is: our page is making one more HTTP request than it *truly* needs to. If you think about it... that's kind of a performance "bug": we're making 2 HTTP requests and we only need 1.

In an ideal world, when we find a bug, the process for fixing it looks like this. First, write a test for the *expected* behavior. Second, run that test and watch it fail. And third, fix the bug and make sure the test passes.

Whelp, when it comes to a *performance* bug... we can do the *exact* same thing! We can write a functional test that *asserts* that this endpoint only makes *one* HTTP request. It's... pretty awesome.

Running the Functional Test

Find your editor and open tests/Controller/MainControllerTest.php. I already set up a functional test that makes a request to /api/github-organization and checks some basic data on the response:

19 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 2
namespace App\Tests\Controller;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
class MainControllerTest extends WebTestCase
{
public function testGetGitHubOrganization()
{
$client = static::createClient();
$client->request('GET', '/api/github-organization');
$this->assertResponseIsSuccessful();
$data = json_decode($client->getResponse()->getContent(), true);
$this->assertArrayHasKey('organization', $data);
}
}
```

Let's makes sure this passes. Run PHPUnit and point it directly at this class:

php bin/phpunit tests/Controller/MainControlerTest.php

The first time you run this script, it will probably download PHPUnit in the background. When it finishes... go tests go! All green.

Adding a Performance Assertion

Here's the idea: in addition to asserting that this response contains JSON with an organization key, I *also* want to assert that it only made one HTTP request. To do that, first add a trait from the SDK: use TestCaseTrait. Next, in the method, add $blackfireConfig = new Configuration() - the one from Blackfire\Profile: the *same* Configuration class we used earlier when we gave our custom-created profile a title. This time call assert() and pass it a *very* special string: metrics.http.requests.count == 1:

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 5
use Blackfire\Profile\Configuration;
... lines 7 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
$client = static::createClient();
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
... lines 19 - 26
}
}
```

I'll show you where that came from soon. Finally, below this, call $this->assertBlackfire() and pass this $blackfireConfig and a callback function:

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 5
use Blackfire\Profile\Configuration;
... lines 7 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
... lines 15 - 16
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
$this->assertBlackfire($blackfireConfig, function() use ($client) {
... lines 21 - 25
});
}
}
```

So... this confused me at first. When we call $this->assertBlackfire() it will execute this callback. Inside, we will do whatever work we want - like making the request. Finally, when the callback finishes, Blackfire will execute this assertion against the code that we ran:

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 5
use Blackfire\Profile\Configuration;
... lines 7 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
... lines 15 - 16
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
$this->assertBlackfire($blackfireConfig, function() use ($client) {
$client->request('GET', '/api/github-organization');
$this->assertResponseIsSuccessful();
$data = json_decode($client->getResponse()->getContent(), true);
$this->assertArrayHasKey('organization', $data);
});
}
}
```

To get this to work, we need to use ($client):

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
... lines 15 - 19
$this->assertBlackfire($blackfireConfig, function() use ($client) {
... lines 21 - 25
});
}
}
```

If this doesn't make sense yet... don't worry: we'll dive a bit deeper soon. But right now... try it! Run the test again:

php bin/phpunit tests/Controller/MainControlerTest.php

And... it fails! Woo! Failed that metrics.http.requests.count == 1!

Performance Tests Create Real Profiles

Behind the scenes, the Blackfire SDK created a *real* Blackfire profile for the request! You can even copy the profile URL and go check it out! This takes us to an "assertions" tab. We're making 2 requests instead of the expected one. We'll talk a lot more about assertions soon.

Ok, but how did this *really* work? It's beautifully simple. When you run the test, it *does* make a real Blackfire profile in the background. However, if you go to your Blackfire homepage, you won't see it.

Why? Hold Cmd or Ctrl and click the assertBlackfire() method. I love it: this method uses the SDK - *just* like we did! - to create a *real* profile. When it does that, it *also* adds a skip_timeline option, which simply tells Blackfire to hide this from our profile page... so it doesn't get cluttered up with all these test profiles. You can *totally* override that if you wanted... via the Configuration object.

In reality, the Blackfire PHPUnit integration is doing the *exact* same thing that we just finished doing in our code: manually creating a new profile. This is *really* nothing new... and I *love* that!

Except... for this metrics thing. Where did that string come from? And what else can we do here? Let's dive into metrics next.

Chapter 23: All about Metrics

Where did this metrics string come from - this metrics.http.requests.count?

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
... lines 15 - 16
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
... lines 19 - 26
}
}
```

There are two things I want to say about this. First, Blackfire stores *tons* of raw data about your profile in little "categories" called metrics. More on that soon. And second, inside the assert() call, you're using a special "expression" language that's similar to JavaScript. It's technically Symfony's ExpressionLanguage if you want to read more. Behind-the-scenes, metrics is probably some object... and we're referencing an http property, then a requests... property then a count property & then we're comparing that to 1.

What Metrics are Available

Ok, cool. So... how the heck did I know to use this *exact* string to get the HTTP call count? This goes back to the Blackfire timeline. On the profile, click the timeline link.

When we talked about the timeline earlier, we talked about how, on the left side, there are these "timeline" metrics. At *that* point, these were just a nice way to add color to different sections of the timeline.

But *now* we understand that there is a *lot* more power behind this info: this shows us *all* the pieces of data we can use in our tests... and in other places that we'll talk about soon.

For example, there's a metric called symfony.events.count which equals seven. You could use that in a metric if, for some reason, you wanted to assert that a certain number of events were dispatched. If I needed to do an assertion about the number of HTTP requests, I would probably search the metrics for http. Apparently there are two... and if you looked closer, you'd find that http.requests is *perfect*. Most of these metrics have data about multiple *dimensions*: we can say http.requests.count to get the actual number or http.requests.memory to get how much memory they used.

In the test system, we start with metrics. then use *anything* we find here.

Fixing the Performance Bug

We now have a performance bug in our application that we've *proven* with a test. And at this point, the actual way we *fix* that bug is not as important: all we care about is that we can change some code and get this test to pass.

The logic for the API calls lives in src/GitHub/GitHubApiHelper.php: it has two public function and each makes one API request:

49 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 15
public function getOrganizationInfo(string $organization): GitHubOrganization
{
$response = $this->httpClient->request('GET', 'https://api.github.com/orgs/'.$organization);
... lines 19 - 25
}
... lines 27 - 30
public function getOrganizationRepositories(string $organization): array
{
$response = $this->httpClient->request('GET', sprintf('https://api.github.com/orgs/%s/repos', $organization));
... lines 34 - 46
}
}
```

How can we make this page only make *1* HTTP request? Well, if you looked closely.. Ah! Too close! Ahh. You'd find that you can get all the information you need by *only* making this *second* HTTP request. The details aren't important - so let's just jump in.

Add a new property called $githubOrganizations set to an empty array:

68 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 10
private $githubOrganizations = [];
... lines 12 - 66
}
```

As we loop over the repositories for a specific organization, we will *store* that organization's info. Add a new variable called $publicRepoCount set to 0:

68 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 10
private $githubOrganizations = [];
... lines 12 - 37
public function getOrganizationRepositories(string $organization): array
{
... lines 40 - 44
$publicRepoCount = 0;
foreach ($data as $repoData) {
... lines 47 - 55
}
... lines 57 - 65
}
}
```

the number of public repositories an organization has is one of the pieces of data we need.

Then, inside the foreach: if $repoData['private'] === false - that's one of the keys on $repoData - say ++$publicRepoCount:

68 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 10
private $githubOrganizations = [];
... lines 12 - 37
public function getOrganizationRepositories(string $organization): array
{
... lines 40 - 44
$publicRepoCount = 0;
foreach ($data as $repoData) {
... lines 47 - 52
if ($repoData['private'] === false) {
++$publicRepoCount;
}
}
... lines 57 - 65
}
}
```

So, as we're looping over the repositories, we're counting how many are public.

Finally, at the bottom, if *not* isset($this->githubOrganizations[$organization]), then $this->githubOrganizations[$organization] = new GitHubOrganization():

68 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 10
private $githubOrganizations = [];
... lines 12 - 37
public function getOrganizationRepositories(string $organization): array
{
... lines 40 - 44
$publicRepoCount = 0;
foreach ($data as $repoData) {
... lines 47 - 52
if ($repoData['private'] === false) {
++$publicRepoCount;
}
}
if (!isset($this->githubOrganizations[$organization])) {
$this->githubOrganizations[$organization] = new GitHubOrganization(
... lines 60 - 61
);
}
... lines 64 - 65
}
}
```

This needs two arguments. The first is the organization name. We can probably use the $organization argument... or you can use $data[0] - to get the first repository - then ['owner']['login']. For the second argument, pass $publicRepoCount:

68 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 10
private $githubOrganizations = [];
... lines 12 - 37
public function getOrganizationRepositories(string $organization): array
{
... lines 40 - 44
$publicRepoCount = 0;
foreach ($data as $repoData) {
... lines 47 - 52
if ($repoData['private'] === false) {
++$publicRepoCount;
}
}
if (!isset($this->githubOrganizations[$organization])) {
$this->githubOrganizations[$organization] = new GitHubOrganization(
$data[0]['owner']['login'],
$publicRepoCount
);
}
... lines 64 - 65
}
}
```

Now, *each* time we call this method, we *capture* the organization's information and store it on this property. So if we call this method *first* and *then* the other method... we could *cheat* and return the GitHubOrganization object that's stored on the property. It's property caching!

Check it out: if isset($this->githubOrganizations[$organization]) then return that immediately without doing any work:

68 lines src/GitHub/GitHubApiHelper.php

```
... lines 1 - 6
class GitHubApiHelper
{
... lines 9 - 17
public function getOrganizationInfo(string $organization): GitHubOrganization
{
// optimization in case getOrganizationRepositories is called first
if (isset($this->githubOrganizations[$organization])) {
return $this->githubOrganizations[$organization];
}
... lines 24 - 32
}
... lines 34 - 66
}
```

So... *are* we calling these two methods in the "correct" order to get this to work? Check out the controller:

122 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 54
public function gitHubOrganizationInfo(GitHubApiHelper $apiHelper)
{
... line 57
$organization = $apiHelper->getOrganizationInfo($organizationName);
$repositories = $apiHelper->getOrganizationRepositories($organizationName);
... lines 60 - 64
}
... lines 66 - 120
}
```

Nope! Swap these two lines so the *first* call will set up the caching for the second:

122 lines src/Controller/MainController.php

```
... lines 1 - 16
class MainController extends AbstractController
{
... lines 19 - 54
public function gitHubOrganizationInfo(GitHubApiHelper $apiHelper)
{
... line 57
$repositories = $apiHelper->getOrganizationRepositories($organizationName);
$organization = $apiHelper->getOrganizationInfo($organizationName);
... lines 60 - 64
}
... lines 66 - 120
}
```

Phew! Let's see if that helps. It was a complicated fix... but thanks to our test, we will know for *sure* if it worked. Go!

php bin/phpunit tests/Controller/MainControlerTest.php

They pass! This *proves* that we reduced the HTTP calls from two to one.

Typos in Metrics

What I *love* about the metrics system is that there are *many* to choose from. What I *don't* love is that you need to manually look up everything that's available. *Fortunately*, if you make a typo - the error is great. Change count to vount:

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
... lines 15 - 16
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.vount == 1');
... lines 19 - 26
}
}
```

And re-run the test:

php bin/phpunit tests/Controller/MainControlerTest.php

> An error occurred when profiling the test

And when we follow the profile link... check out that error!

> The following assertions are not valid... Property "vount" does not exist, available ones are:

... and it lists all the properties. That's *super* friendly. Fix the typo:

29 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
... lines 15 - 16
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
... lines 19 - 26
}
}
```

Organizing Blackfire Assertions into Separate Test Cases

The *one* downside to adding Blackfire assertions in your tests is that they *do* slow things down because instrumentation happens and we need to wait for Blackfire to create the profile.

Because of that, as a best practice, we usually like to isolate our performance tests from our normal tests. Check it out: copy the test method name, paste it below, and call it testGetGitHubOrganizationBlackfireHttpRequests():

41 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 28
public function testGetGitHubOrganizationBlackfireHttpRequests()
{
... lines 31 - 38
}
}
```

And... copy the contents of the original method and paste here. Now... we only need to create the $client, create $blackfireConfig and, inside assertBlackfire(), *just* make the request:

41 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 28
public function testGetGitHubOrganizationBlackfireHttpRequests()
{
$client = static::createClient();
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
$this->assertBlackfire($blackfireConfig, function() use ($client) {
$client->request('GET', '/api/github-organization');
});
}
}
```

Back in the original method, we can simplify... in fact we can go *all* the way back to the way it was before: create the client, make the request, assert something:

36 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 12
public function testGetGitHubOrganization()
{
$client = static::createClient();
$client->request('GET', '/api/github-organization');
$this->assertResponseIsSuccessful();
$data = json_decode($client->getResponse()->getContent(), true);
$this->assertArrayHasKey('organization', $data);
}
... lines 23 - 34
}
```

Why is this useful? Because *now* we can *skip* the Blackfire tests if we're just trying to get something to work. How? Above the performance test, add @group blackfire:

40 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 23
/**
* @group blackfire
... line 26
*/
public function testGetGitHubOrganizationBlackfireHttpRequests()
{
... lines 30 - 37
}
}
```

Thanks to that, we can add --exclude-group=blackfire to *avoid* the Blackfire tests:

php bin/phpunit tests/Controller/MainControlerTest.php --exclude-group=blackfire

Yep! Just one test, two assertions. Another nice detail is to add @requires extension blackfire:

40 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 23
/**
... line 25
* @requires extension blackfire
*/
public function testGetGitHubOrganizationBlackfireHttpRequests()
{
... lines 30 - 37
}
}
```

Now, if someone is *missing* the Blackfire extension, instead of the tests exploding, they'll be marked as skipped.

Don't do Time-Based Assertions

The *last* thing I want to mention about assertions is this: please, please *please* avoid time-based assertions. They're the *easiest* to create - I know. It's *super* tempting to want to create an assertion that the request should take less than 500 milliseconds. If you *do* this, you will hate your tests.

Why? Because there's *way* too much variability in time: the request might run fast enough on one machine, but *not* fast enough on another. Or your server *might* just have a bad day... and suddenly your tests are failing. Relying on time makes your tests fragile.

Next, we're going to talk *more* about metrics and assertions. We know that we can add assertions to profiles that are created inside our tests.

But we an *also* add *global* assertions: tests that run *any* time you create a profile for *any* page! If you want to make sure that a specific page - or *any* page - doesn't make more than, I don't know, 10 database queries, you can add an "assertion" for that and see a *big* failure if you break the rules.

Chapter 24: Assertions / Profile "Tests"

Adding specific assertions inside a test is really cool:

40 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 27
public function testGetGitHubOrganizationBlackfireHttpRequests()
{
... lines 30 - 31
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
... lines 35 - 37
}
}
```

But you can *also* add assertions *globally*. What I mean is, whenever you trigger a *real* Blackfire profile - like through your browser - you can set up *assertions* that you want to run against that profile.

Recommendations Versus Assertions

Actually, we've *already* seen a system that's *similar* to this. Click into one of the profiles. Every profile has a "Recommendations" tab on the left, which tells us changes that we should *probably* make. In reality, recommendations are *assertions* in disguise! For example, the "Symfony debug mode should be disabled in production" is displayed here because the *assertion* that metrics.symfony.kernel.debug.count equals zero, failed. Yep, metrics are *everywhere*!

I *love* that Blackfire gives us so many of these recommendations for free. But we can *also* define our own. When we do, they'll show up under the assertions tab.

Hello .blackfire.yaml

How do we do that? Just send an email to assertion-requests@blackfire.io, pay $19.95 for shipping and handling, and wait 6-8 weeks for delivery. If you order now, we'll *double* your order and include a signed-copy of the blackfire-player source code printed as a book.

Or you can configure global assertions with a special Blackfire config file. At the root of your project, create a new file called .blackfire.yaml. A few different things will eventually go here - the first is tests:.

Honestly, the *trickiest* thing about writing assertions is trying to figure out... a good assertion to use! Writing *time-based* assertions is the easiest... but because they're fragile, we want to avoid those.

Adding your first "Test"

Let's start with one we've already done. Say: "HTTP requests should be limited to 1 per page":. Below this, add path set to the regular expression /.*:

6 lines .blackfire.yaml

```
"tests":
"HTTP Requests should be limited to 1 per page":
path: "/.*"
... lines 4 - 6
```

This means that this assertion will be executed against *any* profile for *any* page. Only want the assertion to run against a single page or section? Use this option.

Now add assertions: with one item below. Go steal the metrics expression from our test... and paste it here. Change this to be *less than* or equal to 1:

6 lines .blackfire.yaml

```
"tests":
"HTTP Requests should be limited to 1 per page":
path: "/.*"
assertions:
- "metrics.http.requests.count
```

That's it! Let's try it out! Back in your browser... go back to our site, refresh, and create a new profile. I'll call it: [Recording] Added first assertion.

Click into the call graph. Actually, go back. See this little green check mark? That *already* tells us that this profile passed *all* our "tests". We can see that on

the "Assertions" tab: metrics.http.requests.count was 0, which *is* less than or equal to 1.

So at this point, these "tests" are basically a nice way to create your *own* custom recommendations. These will become *more* interesting later when we talk about environments and builds.

Next, let's talk about a tool from the Blackfire ecosystem called the Blackfire player. It's a command line utility that allows us to write simple files and execute them as functional tests... *completely* independent of the Blackfire profiling system. What we learn from it will form the foundation for the rest of the tutorial.

Chapter 25: Blackfire Player

Pretend for a few minutes that the Blackfire profiler that we've been learning *so* much about... doesn't exist... at all. Why? Because we're *now* going to talk about something that has the word "Blackfire" in it... but has absolutely *nothing* to do with the Blackfire profiler. At least, not yet.

Hello Blackfire Player

Google for "Blackfire player". The Blackfire Player is an open source library that makes it *really* easy to write a few lines of code that will then be executed to *crawl* a site: clicking on links, filling out forms, and doing things with the result. It's basically a simple language for surfing the web and a tool that's able to *read* that language and... actually do it!

To install it, copy the curl command, find your terminal, and paste:

curl -OLsS https://get.blackfire.io/blackfire-player.phar

If you're on Windows, you can just download the blackfire-player.phar file from that URL and put it into your project.

Now go back and copy the other two commands.

chmod +x blackfire-player.phar
mv blackfire-player.phar /usr/local/bin/blackfire-player

Paste and... that's it! For Windows users, skip this step. Let's see if it works. Run:

blackfire-player

Nice!

Tip

For Windows, run php blackfire-player.php from inside your project.

So here's the idea: we create a file that contains one or more *scenarios*. Inside each scenario, we write code that says: go visit this URL, expect a 200 status code, then click on this link, and so on. It can get fancier, but that's the gist of it.

Creating our First Scenario & .bkf File

Let's create a our first Blackfire player file at the root of the project, though it could live anywhere. Call it, how about, scenario.bkf. That's *pure* creativity.

At the top, I'll put a name - though it's not very important - then endpoint set to our server's URL. So https://localhost:8000:

5 lines scenario.bkf

```
name "Various scenarios for the site"
# override with --endpoint option
endpoint "https://localhost:8000"
```

You can override this when you *execute* this file by passing a --endpoint option.

Notice that this *kind* of looks like YAML, but it's *not*: there is no : between the key and value. This is a custom Blackfire player language, which is friendly, but takes some getting used to.

At the bottom, add our first scenario - call it "Basic Visit". Inside, let's do two things: first, visit url("/"). We can *also* give this page a name - it helps debugging:

14 lines scenario.bkf

```
name "Various scenarios for the site"
# override with --endpoint option
endpoint "https://localhost:8000"
scenario
name "Basic Visit"
visit url("/")
name "Homepage"
... lines 11 - 14
```

And second... once we're on the homepage, let's "click" this "Log In" link. Do that with click link() and then use that exact text: Log In. Give this page a name too:

14 lines scenario.bkf

```
name "Various scenarios for the site"
# override with --endpoint option
endpoint "https://localhost:8000"
scenario
name "Basic Visit"
visit url("/")
name "Homepage"
click link("Log In")
name "Login page"
```

Executing blackfire-player

That's enough to start. We *should* be able to use the blackfire-player tool to... actually *do* this stuff!. Let's try it:

blackfire-player run scenario.bkf

And... it fails:

> Curl error 60...

If you Google'd this, you find out that this is an SSL problem - it's caused because or Symfony dev server uses a, sort of, self-signed certificate that blackfire-player doesn't like. The simplest solution, which is ok since we're just testing locally - is to pass --ssl-no-verify

blackfire-player run scenario.bkf --ssl-no-verify

And... hey! It worked! Scenarios 1, steps 2. It *truly* made a request to the homepage then clicked on that link! By the way, the requests aren't using a *real* browser. And so, any JavaScript code on your page *won't* run. That *might* change in the future - but I'm not sure.

Anyways, to see more fun output, use the -v flag:

blackfire-player run scenario.bkf --ssl-no-verify -v

Very cool! Blackfire player *is* now making two real HTTP requests to our site... but it's not *doing* anything with that data. Next, let's add some *tests* to our scenario - like expecting that the status code is 200 and checking for elements in the DOM.

Chapter 26: Expectations/Tests with Blackfire Player

We just used blackfire-player to execute our first "scenario". It's pretty simple: it goes to the homepage then clicks the "Log In" link:

14 lines scenario.bkf

```
name "Various scenarios for the site"
# override with --endpoint option
endpoint "https://localhost:8000"
scenario
name "Basic Visit"
visit url("/")
name "Homepage"
click link("Log In")
name "Login page"
```

It works... but... we're not *doing* anything after we visit these pages. The *true* power of blackfire-player is that you can add *tests* to your scenario - or even scrape pages and save that data somewhere.

Adding an Expectation/Test to a Page

To add a "test" - or "assertion", or "expectation"... I *love* when things have 5 names... - say expect followed by - you guessed it! - an *expression*! status_code() == 200. Copy that and add it to the login page as well:

17 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
name "Homepage"
expect status_code() == 200
click link("Log In")
name "Login page"
expect status_code() == 200
... lines 16 - 17
```

Ok, try blackfire-player again!

blackfire-player run scenario.bkf --ssl-no-verify -v

Woo! It still passes and *now* it's starting to be useful!

What's Possible in the expect Expression?

Let's break this down. First, *just* like we saw with the metrics stuff:

40 lines tests/Controller/MainControllerTest.php

```
... lines 1 - 8
class MainControllerTest extends WebTestCase
{
... lines 11 - 27
public function testGetGitHubOrganizationBlackfireHttpRequests()
{
... lines 30 - 31
$blackfireConfig = (new Configuration())
->assert('metrics.http.requests.count == 1');
... lines 34 - 37
}
}
```

This is an *expression* - it's Symfony's ExpressionLanguage once again - basically JavaScript. And second... this expression has a *ton* of built-in functions.

Search the blackfire-player docs for "status_code"... and keep searching until you find a big function list. Here it is. Yep, we can use current_url(), header() to get a header value and many others. The css() function is especially useful: it allows us to dig into the HTML on the page. We'll use that in a minute. The docs also have good examples of how to do more complex things. But we're not going to become Blackfire player experts right now... I just want you to get

comfortable with writing scenarios.

Asserting HTML Elements with css()

Let's try to write a *failing* expectation to see what it looks like. Let's see... we could find this table and assert that it has more than 500 rows... which it definitely does *not*. Let's find a CSS selector we can use... hmm. Ok, we could look for a <tbody> with this js-sightings-list class and then count its <tr> elements.

Back inside the scenario file, add another expect. This time use the css() function and pass it a CSS selector: tbody.js-sightings-list tr:

18 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
name "Homepage"
expect status_code() == 200
expect css("tbody.js-sightings-list tr").count() > 500
... lines 13 - 18
```

Internally, The blackfire-player uses Symfony's Crawler object from the DomCrawler component, which has a count() method on it. Assert that this is > 500.

Let's see what happens!

blackfire-player run scenario.bkf --ssl-no-verify -v

And... yes! It fails - with a nice error:

> The count() of that CSS element is 25, which is not greater than 500.

Go back and change this to 10:

18 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
... lines 10 - 11
expect css("tbody.js-sightings-list tr").count() > 10
... lines 13 - 18
```

The data is dynamic data... so we don't *really* know how many rows it will have. But since our fixtures add more than 10 sightings... and because there will probably be at least 10 sightings if we ever ran this against production, this is probably a safe value.

Try it now:

blackfire-player run scenario.bkf --ssl-no-verify -v

All better!

Typos in Expressions

Another thing that blackfire-player does well is its *errors* when I... do something silly. Make a typo: change count() to ount():

18 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
... lines 10 - 11
expect css("tbody.js-sightings-list tr").ount() > 10
... lines 13 - 18
```

And rerun the scenario:

blackfire-player run scenario.bkf --ssl-no-verify -v

> Unable to call method ount of object Crawler.

That's a *huge* hint to tell you what object you're working with so you can figure out what methods it *does* have. Change that back to count():

18 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
... lines 10 - 11
expect css("tbody.js-sightings-list tr").count() > 10
... lines 13 - 18
```

Performance Assertions in the Scenarios?

So... blackfire-player has *nothing* to do with the Blackfire profiler. It's just a useful tool for visiting pages, clicking on links and adding expectations. But... if it *truly* had nothing to do with the profiler, I probably wouldn't have talked about it. In reality, the concept of "scenarios" is *about* to become *very* important - it's a fundamental part of a topic we'll talk about soon: Blackfire "builds".

And actually, there is one *little* integration between blackfire-player and the profiler: you can add *performance* assertions to your scenario. To do that, instead of expect, say assert and then use any performance expression you want: the same strings that you can use inside a test. For example: metrics.sql.queries.count < 30:

20 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
... lines 10 - 13
assert metrics.sql.queries.count < 30
... lines 15 - 20
```

When we execute this:

blackfire-player run scenario.bkf --ssl-no-verify -v

It *does* still pass. But if you played with this value - like set it to < 1 and re-ran the scenario:

blackfire-player run scenario.bkf --ssl-no-verify -v

Hmm, it *still* passes... even though this page is *definitely* making more than one query. The reason is that the assert functionality *won't* work inside a scenario until we introduce Blackfire "environments" - which we will soon. They are one of my absolute *favorite* parts of Blackfire.

For now, I'll leave a comment that this *won't* work until then:

20 lines scenario.bkf

```
... lines 1 - 5
scenario
... lines 7 - 8
visit url("/")
... lines 10 - 12
# won't work until we're using Blackfire environment
assert metrics.sql.queries.count < 30
... lines 15 - 20
```

Next, let's deploy to production! Because once our site is deployed, we can *finally* talk about cool things like "environments" and "builds". You can use anything to deploy, of course, but *we* will use SymfonyCloud.

Chapter 27: Deploying to SymfonyCloud

Transition point! *Everything* we've talked about so far has included profiling our *local* version of the site. But things get even *cooler* when we start to profile our *production* site. Having *real* data often shows performance problems that you just *can't* anticipate locally. And because of the way that Blackfire works, we can create profiles on production *without* slowing down our servers and affecting real users. *Plus*, once we're profiling on production, we can unlock even *more* Blackfire features.

So... let's get this thing deployed! You can use *any* hosting system you want, but I'm going to deploy with SymfonyCloud: it's what we use for SymfonyCasts and it makes deployment dead-simple for Symfony apps. It also has a free trial if you want to code along with me.

Initializing your SymfonyCloud Project

Find your terminal and make sure you're on your master branch. That's not required, but will make life easier. Start by running:

```
symfony project:init
```

This will create a few config files that tell SymfonyCloud *everything* it needs to know to deploy our site. The most important file is .symfony.cloud.yaml:

39 lines .symfony.cloud.yaml

```
name: app
type: php:7.1
runtime:
extensions:
- apcu
- mbstring
- ctype
- iconv
build:
flavor: none
web:
locations:
"/":
root: "public"
expires: 1h
passthru: "/index.php"
disk: 512
mounts:
"/var": { source: local, source_path: var }
hooks:
build: |
set -x -e
curl -s https://get.symfony.com/cloud/configurator | (>&2 bash)
(>&2 symfony-build)
deploy: |
set -x -e
(>&2 symfony-deploy)
```

Ah, this says we want PHP 7.1. Let's *upgrade* by changing that to 7.3:

39 lines .symfony.cloud.yaml

```
... lines 1 - 2
type: php:7.3
... lines 4 - 39
```

Back at the terminal, copy the *big* git command: this will add all the new files to git and commit them:

```
git add .symfony.cloud.yaml .symfony/services.yaml .symfony/routes.yaml php.ini
git commit -m "Add SymfonyCloud configuration"
```

Next, to *tell* SymfonyCloud that we want a new "server" on their system, run:

```
symfony project:create
```

Every "site" in SymfonyCloud is known as a "project" and we only need to run this command *once* per app. You can ignore the big yellow warning - that's

because I have a few other SymfonyCloud projects attached on my account. Let's call the project "Sasquatch Sightings" - that's just a name to help us identify it - and choose the "Development" plan.

The development plan includes a free 7 day trial... which is *awesome*. You *do* need to enter your credit card info - that's a way to prevent spammers from creating free trials - but it won't be charged unless you run symfony project:billing:accept later to keep this project permanently.

I already have a credit card on file, so I'll use that one. Once we confirm, this *provisions* our project in the background... I *assume* it's waking up thousands of friendly robots who are carefully creating our new space in... the "cloud". Hey! There's one now... dancing!

And... done!

Deploying & Security Checks

Ready for our first deploy? Just type:

symfony app:prepare:deploy --branch=master --confirm --this-is-not-a-real-command

Kidding! Just run:

symfony deploy

And... hello error! This is actually great. Really! The deploy command automatically checks your composer.lock file to see if you're using any dependencies with known security vulnerabilities. Some of my Symfony packages *do* have vulnerabilities... and if this were a real app, I would upgrade those to fix that problem. But... because this is a tutorial... I'm going to ignore this.

Our First Deploy

Run the command again with a --bypass-checks flag:

symfony deploy --bypass-checks

We still see the big message... but it's deploying! This takes care of *many* things automatically, like running composer install and executing database migrations. This first deploy will be slow - especially to download all the Composer dependencies. I'll fast-forward. It also handles setting up Webpack Encore... and even creates a shiny new SSL certificate. Those are *busy* robots!

And... done! It dumped out a funny-looking URL. Copy that. In a *real* project, you will attach your *real* domain to SymfonyCloud. But this "fake" domain will work *beautifully* for us.

Spin back over and pop that URL into your browser to see... a beautiful 500 error! Wah, wah. Actually, we're *super* close to this all working. Next, let's use a special command to debug this error, add a database to SymfonyCloud - yep, that's the piece we're missing - and load some dummy data over a "tunnel". *Lots* of good, nerdiness!

Chapter 28: Database Tricks on SymfonyCloud

We just deployed to SymfonyCloud!!! Well, I mean, we *did*... but it doesn't... ya know... *work* yet. Because this is the *production* 500 error, we can't see the real problem.

No worries! Head back to your terminal. The symfony command has an easy way to check the production logs. It is...

symfony logs

This prints a list of *all* the logs. The app/ directory is where our application is deployed to - so the first item is our project's var/log/prod.log file. You can also check out the raw access log... or everything. Hit 0 to "tail" the prod.log file. And... there it is:

> An exception has occurred... Connection refused.

Adding a Database to SymfonyCloud

I recognize this: it's a database error.... which... hmm... makes sense: we haven't told SymfonyCloud that we *need* a database! Let's go do that!

Google for "SymfonyCloud MySQL" to find... oh! A page that talks about *exactly* that. Ok, we need to add a little bit of config to 2 files. The first is .symfony/services.yaml. This is where you tell SymfonyCloud about all the "services" you need - like a database service, ElasticSearch, Redis, RabbitMQ, etc.

Copy the config for .symfony/services.yaml... then open that file and paste:

5 lines .symfony/services.yaml

```
mydatabase:
# mariadb
type: mysql:10.2
disk: 1024
```

The database is actually MariaDB, which is why the version here is 10.2: MariaDB version 10.2.

Notice that we've used the key mydatabase. That can be *anything* you want: we'll *reference* this string from the *other* config file that we need to change: .symfony.cloud.yaml.

Inside *that* file, we need a relationships key: this is what *binds* the web container to that database service. Let's see... we don't have a relationships key yet, so let's add it: relationships and, below, add our *first* relationship with a special string: database set to mydatabase:mysql:

42 lines .symfony.cloud.yaml

```
... lines 1 - 24
relationships:
database: "mydatabase:mysql"
... lines 27 - 42
```

This syntax... is a little funny. The mydatabase part is referring to whatever key we used in services.yaml - and then we say :mysql... because that service is a mysql type.

The *really* important thing is that we called this relationship database. Thanks to that SymfonyCloud will expose an environment variable called DATABASE_URL which contains the *full* MySQL connection string: username, host, database name and all:

29 lines .env

```
... lines 1 - 26
DATABASE_URL=mysql://root:@127.0.0.1:3306/blackfire
... lines 28 - 29
```

It's literally DATABASE_URL and not PIZZA_URL because we called the relationship database instead of pizza... which would have been less descriptive, but more delicious.

This is important because DATABASE_URL *happens* to be the environment variable that our app will use to connect to the database. In other words, our app will *instantly* have database config.

Back at the terminal, hit Ctrl+C to exit from logging. Let's add the two changes and commit them:

git add .
git commit -m "adding SfCloud database"

Now, deploy!

symfony deploy

Oh, duh - run with the --bypass-checks flag:

symfony deploy --bypass-checks

The deploy will still take some time - it has a lot of work to do - but it'll be faster than before. When it finishes... it dumps the same URL - that won't change. But to be even *lazier* than last time, let's tell the command to open this URL in my browser... *for* me:

symfony open:remote

Tunneling to the Database

And... we have a deployed site! Woo! The database is empty... but if this were a real app, it would start to be populated by *real* users entering their *real* Bigfoot sightings... cause Bigfoot is... totally real.

But... to make this a bit more interesting for *us*, let's load the fixture data one time on production.

This is a bit tricky because the fixture system - which comes from DoctrineFixturesBundle - is a Composer "dev" dependency... which means that it's not even *installed* on production. That's good for performance. If it *were* installed, we could run:

symfony ssh

To SSH into our container, and then execute the command to load the fixtures. But... that won't work.

No problem! We can do something cooler. Exit out of SSH, and run:

symfony tunnel:open

I *love* this feature. Normally, the remote database isn't accessible by *anything* other than our container: you can't connect to it from anywhere else on the Internet. It's totally firewalled. But *suddenly*, we can connect to the production database locally on port 30000. We can *use* that to run the fixtures command locally - but send the data up to *that* database. Do it by running:

DATABASE_URL=mysql://root:@127.0.0.1:30000/main php bin/console doctrine:fixtures:load

Ok, let's break this down. First, there is actually a *much* easier way to do all of this... but I'll save that for some future SymfonyCloud tutorial. Basically, we're running the doctrine:fixtures:load command but sending it a *different* DATABASE_URL: one that points at our production database. When you open a tunnel, you can access the database with root user, no password - and the database is called main.

The only problem is that this command... takes *forever* to run. I'm not sure exactly why - but it *is* doing all of this over a network. Go grab some coffee and come back in a few minutes.

When it finishes... yes! Go refresh the page! Ha! We have a production site with at least *enough* data to make profiling interesting.

Next, let's do that! Let's configure Blackfire on production! That's easy right? Just repeat the Blackfire install process on a different server... right? Yep! Wait, no! Yes! Bah! To explain, we need to talk about a *wonderful* concept in Blackfire called "environments".

Chapter 29: Blackfire Environments

Now that our site is *deployed* - woo! - how can we get Blackfire working on it? Well... we already know the answer. If you find the Blackfire Install page... it makes it easy: I want to install on "a server"... and let's pretend it uses Ubuntu.

Getting Blackfire installed on your production machine is as easy as running the commands below to install the Blackfire PHP extension - the Probe, install the Agent and configure the agent with our server id and token. Easy peasy!

Hello: Environments

But.... *some* Blackfire account levels - offer a kick-butt feature called *environments*. If you have access to Blackfire environments - or if you're able to get a "plan" that offers environments, I highly recommend them.

Tip

Blackfire environments require a Premium plan or higher.

An environment is basically an isolated Blackfire account. When you have an environment, you send your *profiles* to that environment. The first advantage is that you can *invite* multiple people to an environment, which means that *anyone* can profile your production site and see other profiles made by people on your team. It also has *other* superpowers - ahem, builds - that *really* make it shine.

Understanding Organizations

So let's create an environment! Go back to https://blackfire.io and click on the "Environments" tab. Actually, click on the "Organizations" tab... that's where this all starts. Blackfire organizations are a bit like GitHub organizations. With GitHub, you can subscribe to a "plan" directly on your *personal* account *or* you can create an organization, have *it* subscribe & pay for a plan, and then invite individual users to the organization. Blackfire organizations work *exactly* like that. And if you want to use environments, you need to create an organization and subscribe to a Blackfire plan *through* that organization.

This *did* confuse me a bit at first. Basically, unless you just want the *lowest* Blackfire paid plan, you should probably *always* create an organization and subscribe to Blackfire through *it*. It just has a few more features than subscribing with your personal account.

Creating an Environment

Anyways, I've already got an organization set up and subscribed to a plan. Once you *have* an organization, you can click into it to create a new environment. I already have one for SymfonyCasts.com production. Click to create a new one. Let's call it: "Sasquatch Sightings Production".

For the "Environment Endpoint", it wants the URL to the site. Again, if this were a *real* project, I would attach a *real* domain... but copy the weird domain name, and paste. Select your timezone, sip some coffee, and... "Create environment"!

On the second step, it asks us to provide URLs to test... and it starts with just one: the homepage. We're going to talk more about this soon, so just leave it. I'll also uncheck the build notifications - more on those later.

Environment vs Personal Server Credentials

Hit "Save settings" and... we're done! It rewards us with a shiny new "Server Id" and "Server Token".

This is *super* important. No matter *how* you install Blackfire on a server, you eventually need to configure the "Server id" and "Server Token". This is *basically* a username & password that tells Blackfire which *account* a profile should be sent to.

When you register with Blackfire, it immediately created a "Server Id" and "Server Token" connected with your *personal* account. We used that when we installed Blackfire on our local machine. But now that we have an environment, it has its *own* Server Id and token. The drop-down on the Install page is allowing us to *choose* which credentials we want to see on this page.

Locally, we should *still* use our *personal* credentials: it keeps things cleaner. But on *production*, we should use the new *environment's* Server Id and Token. The install page gives us all the commands we need *using* those credentials.

Oh, and by the way: if you have a "free" personal account... but are attached to an organization with a paid plan, any profiles you create with your *personal* Server Id and Token will *inherit* the features from that organization's plan. That lets us use our personal credentials locally and *still* get all the Blackfire features we're paying for. One exception to that rule, unfortunately, is "Add-Ons".

Configuring Blackfire on SymfonyCloud

Ok, let's get our production machine set up. I'll select "Symfony Cloud" as my host... which takes me to a dedicated page on this topic.

Let's see... step one is, instead of installing Blackfire with something like apt-get, we'll add a line to .symfony.cloud.yaml. I already have an extensions key... so just add blackfire:

42 lines .symfony.cloud.yaml

```
... lines 1 - 4
runtime:
extensions:
... lines 7 - 10
- blackfire
... lines 12 - 42
```

Boom! Blackfire is installed. Add this file to Git... and commit it:

```
git add .
git commit -m "adding blackfire extension"
```

The *other* step is to *configure* Blackfire. Once again, it has a drop-down to select between my personal credentials and credentials for an enivornment. Select our "Sasquatch production" environment. Cool! This gives us a command to set two SymfonyCloud *variables*. Copy that, move over, and paste:

```
symfony var:set BLACKFIRE_SERVER_ID=XXXXXX BLACKFIRE_SERVER_TOKEN=XXXXXX
```

Ok... we're good! To make both changes take effect, deploy!

```
symfony deploy --bypass-checks
```

I'll fast-forward. Once this finishes... move over and refresh. Ok... everything still works. *Now*, moment of truth: open the Blackfire browser extension and create a new profile. It's working! I'll call it: [Recording] First profile in production.

Next, let's... *look* at this profile! It will contain a *few* new things and some data that is much more relevant now that we're on production.

Chapter 30: Production Profile: Cache Stats & More Recommendations

We just profiled our *first* page on production, which is using the Blackfire Server Id and Token for the *environment* we created.

Profiles Belong to the Environment

Go to https://blackfire.io, click "Environments", open our new environment... and click the "Profiles" tab. Yep! Whenever *anyone* creates a profile using this environment's credentials, it will now show up *here*: the profile *belongs* to this environment. We haven't invited any other users to this environment yet, but if we did, they would *immediately* be able to access this area *and* trigger new profiles with *their* browser extension.

If you go to back to https://blackfire.io to see your dashboard, the new profile *also* shows up here. But that's purely for convenience. The profile *truly* belongs to the environment. You can even see that right here. But Blackfire places *all* profiles that *I* create on this page... to make life nicer.

Click the profile to jump into it. Of course... this looks *exactly* like any profile we created on our local machine. But it *does* have a few differences.

Caching Information

Hover over the profile name to find... "Cache Information". We talked about this earlier: it shows stats about *various* different caches on your server and how much space each has available. Now that we're profiling on production, this data is *super* valuable!

For example, if your OPcache filled up, your site would start to slow down *considerably*... but it might not be very obvious when that happens. It's not like there are alarms that go off once PHP runs out of OPcache space. But thanks to this, you can easily see how things *really* look, right now, on production. If any of these are full or nearly full, you can read documentation to see which setting you need to tweak to make that cache bigger.

Quality & Security Recommendations

The *other* thing I want to show you is under "Recommendations" on the left. There are 3 *types* of recommendations... and we have one of each: the first is a *security* recommendation, the second is a *quality* recommendation and the third a *performance* recommendation. Only the *performance* recommendations come standard: the other two require an "Add on"... which I didn't have until I started using my organization's plan.

As always, to get a *lot* more info about a problem and how to fix it, you can click the question mark icon.

Converting Recommendations into Assertions

One of my *favorite* things about recommendations is that you can easily *convert* any of these into an *assertion*. If you click on assertions, you'll remember that we created one "test" that said that every page should have - at *maximum* - one HTTP request.

We configured that inside of our .blackfire.yaml file: we added tests, configured this test to apply to every URL, and leveraged the metrics system to write an expression.

Back on the recommendations, click to see more info on one of these... then scroll down. *Every* recommendation contains code that you can copy into your .blackfire.yaml file to *convert* that recommendation into a *test*... or "assertion".

That *might* not seem important right now... because so far, it looks like doing that would simply "move" this from a "warning" under "Recommendations" to a "failure" under "Assertions"... which is cool... but just a visual difference.

But! In a few minutes, we'll discover that these assertions are *much* more important than they seem. To see why, we need to talk about the *key* feature and superpower of environments: *builds*.

Chapter 31: Automatic Performance Checks: Builds

Head back to https://blackfire.io, click "Environments" and click into our "Sasquatch Sightings Production" environment.

Interesting. By default, it takes us *not* to the profiles tab... but to a tab called "Builds". And, look on the right: "Periodic Builds": "Builds are started every 6 hours"... which we could change to a different interval.

Further below, there are a bunch of "notification channels" where you can tell Blackfire that you want to be *notified* - like via Slack - of the results of this "build" thingy.

Hello Builds

Ok, what the *heck* is a build anyways? To find out, let's trigger one manually, then stand back and see what happens. Click "Start a Build". The form pre-fills the URL to our site... cool... and we can apparently give it a title if we want. Let's... just start the build.

This takes us to a new page where.... interesting: it's running an "Untitled Scenario"... then it looks like it went to the homepage... and created a profile?

Let's... back up: there are a *lot* of interesting things going on. And I *love* interesting things!

First, we've *seen* this word "scenario" before! Earlier, we used the blackfire-player: a command-line tool that's *made* by the Blackfire people... but can be used totally outside of the profiling tool. We created a scenario.bkf file where we defined a *scenario* and used the special blackfire-player language to tell it to go to the homepage, assert a few things, then click on the "Log In" link and check something else:

20 lines scenario.bkf

```
name "Various scenarios for the site"
# override with --endpoint option
endpoint "https://localhost:8000"
scenario
name "Basic Visit"
visit url("/")
name "Homepage"
expect status_code() == 200
expect css("tbody.js-sightings-list tr").count() > 10
# won't work until we're using Blackfire environment
assert metrics.sql.queries.count < 30
click link("Log In")
name "Login page"
expect status_code() == 200
... lines 19 - 20
```

At that time, this was a nice way to "crawl" a site and test some things on it. The "build" used the same "scenario" word. That's not an accident. More on that soon.

Build "URLs to Test"

The *second* important thing is that this profiled the *homepage* because, when we created our environment, we configured one "URL to test": the homepage. That's what the build is doing: "testing" - meaning *profiling* - that page.

Let's add a second URL. One other page we've been working on a lot is /api/github-organization: this JSON endpoint. Copy that URL and add it as a *second* "URL to test". Click save... then manually create a *second* build.

Like before, it creates this "Untitled Scenario" thing. Ah! But *this* time it profiled *both* pages! The build *also* shows up as green: the build "passed".

This is a *critical* thing about builds. It's not *simply* that a build is an automated way to create a profile for a few pages. That would be pretty worthless. The *real* value is that you can write performance *tests* that cause a build to pass or fail.

Check it out "1 successful constraint" - which is that "HTTP Requests should be limited to 1 per page". Hey! That's the "test" that *we* set up inside .blackfire.yaml!

6 lines .blackfire.yaml

```
"tests":
"HTTP Requests should be limited to 1 per page":
path: "/.*"
assertions:
- "metrics.http.requests.count
```

The *real* beauty of tests is *not* that the "Assertions" tab will look red when you're looking inside a profile. The *real* beauty is that you can configure performance *constraints* that should pass *whenever* these builds happen. If a build *fails* - maybe because you introduced some slow code - you can be

notified.

Build Log: blackfire-player

But there's even *more* cool stuff going on. Near the bottom, click to see the "Player output". Woh! It shows us how builds work behind-the-scenes: the Blackfire server *uses* the blackfire-player!

Look closer: it's running a *scenario*: visit url(), method 'GET', then visit url() of /api/github-organization. It's a bit hard to read, but this *converted* our 2 "URLs to test" into a scenario - using the same format as the scenario.bkf file - then *passed* that to blackfire-player. You can even see it *reloading* both pages multiple times to get 10 samples. That's one of the options it added in the scenario.

So with just a *tiny* bit of configuration, Blackfire is now creating a build every 6 hours. Each time, it profiles these 2 pages and, thanks to our one test, if either page makes more than one HTTP request, the build will fail. By setting up a notification, we'll know about it.

The fact that the build system uses blackfire-player makes me wonder: instead of configuring these URLs, could we *instead* have the build system run our custom scenario file? I mean, it's a *lot* more powerful: we can visit pages, but also click links and fill out forms. We can *also* add *specific* assertions to *each* page... in addition to our one "global" test about HTTP requests.

The answer to this question is... of course! And it's where the build system *really* starts to shine. We'll talk about that next.

History & Graphs from Automated Builds

But before we do, I want you to see what the build page looks like once it's had enough time to execute a few automated builds. Let's check out the SymfonyCasts environment. Woh! It's graph time! Because this environment has a *history* of automated builds, Blackfire creates some super cool graphs: like our cache hit percentage and our cache levels. You can see that my OPcache Interned Strings Buffer cache is full. I really need to tweak some config to increase that.

I can also see how the different URLs are performing over time for wall time, I/O, CPU, Memory & network as well as other stuff. We can click to see more details about any build... and even look at any of its profiles.

Anyways, next: let's make the build system smarter by executing our custom scenario.

Chapter 32: Builds with Custom Scenarios

A few chapters ago, we created this scenario.bkf file:

20 lines scenario.bkf

```
name "Various scenarios for the site"
# override with --endpoint option
endpoint "https://localhost:8000"
scenario
name "Basic Visit"
visit url("/")
name "Homepage"
expect status_code() == 200
expect css("tbody.js-sightings-list tr").count() > 10
# won't work until we're using Blackfire environment
assert metrics.sql.queries.count < 30
click link("Log In")
name "Login page"
expect status_code() == 200
... lines 19 - 20
```

It's written in a special blackfire-player *language* where we write one or more "scenarios" that, sort of, "crawl" a web page, asserting things, clicking on links and even submitting forms. This a simple scenario: the tool can do a lot more.

On the surface, apart from its name, this has *nothing* to do with the Blackfire profiler system: blackfire-player is just a tool that can read these scenarios and do what they say. At your terminal, run this file:

blackfire-player run scenario.bkf --ssl-no-verify

That last flag avoids an SSL problem with our local web server. When we hit enter... it goes to the homepage, clicks the "Log In" link and... it passes.

Scenarios in .blackfire.yaml

This is cool... but we can do something *way* more interesting. Copy the entire scenario from this file, close it, and open .blackfire.yaml. Add a new key called scenarios set to a |:

23 lines .blackfire.yaml

```
... lines 1 - 6
scenarios: |
... lines 8 - 23
```

That's a YAML way of saying that we will use multiple lines to set this.

Below, indent, then say #!blackfire-player:

23 lines .blackfire.yaml

```
... lines 1 - 6
scenarios: |
#!blackfire-player
... lines 9 - 23
```

That tells Blackfire that we're about to use the blackfire-player syntax... which is the *only* format supported here... but it's needed anyways. Below, paste the scenario. Make sure it's indented 4 spaces:

23 lines .blackfire.yaml

```
... lines 1 - 6
scenarios: |
#!blackfire-player

scenario
name "Basic Visit"

visit url('/')
name "Homepage"
expect status_code() == 200
expect css("tbody.js-sightings-list tr").count() > 10
# won't work until we're using Blackfire environment
assert metrics.sql.queries.count

click link("Log In")
name "Log in page"
expect status_code() == 200
```

The *cool* thing is that we can *still* execute the scenario locally: just replace scenario.bkf with .blackfire.yaml. The player is smart enough to know that it can look under the scenarios key for our scenarios.

blackfire-player run .blackfire.yaml --ssl-no-verify

But if you run this... error!

> Unable to crawl a non-absolute URI /. Did you forget to set an endpoint?

Duh! Our scenario.bkf file had an endpoint config:

20 lines scenario.bkf

```
... lines 1 - 2
# override with --endpoint option
endpoint "https://localhost:8000"
... lines 5 - 20
```

You *can* copy this into your .blackfire.yaml file. *Or* you can define the endpoint by adding --endpoint=https://localhost:8000:

blackfire-player run .blackfire.yaml --ssl-no-verify --endpoint=https://localhost:8000

Now... it works!

Building the Custom Scenario

So... why did we move the scenario into this file? To find out, add this change to git... and commit it.

```
git add .
git commit -m "moving scenarios into blackfire config file"
```

Then deploy:

```
symfony deploy --bypass-checks
```

Once that finishes... let's go see what changed. First, if we simply went to our site and manually created a profile - like for the homepage - the new scenarios config would have absolutely *no* effect. Scenarios don't do *anything* to an individual profile. Instead, scenarios affect *builds*.

Let's start a new one: I'll give this one a title: "With custom scenarios". Go!

Awesome!! Now, instead of that "Untitled Scenario" that tested the two URLs we configured, it's using our "Basic visit" scenario! It goes to the homepage, then clicks "Log In" to go to that page.

Yep, as *soon* as we add this scenarios key to .blackfire.yaml, it *no longer* tests these URLs. In fact, these are now meaningless. Instead, we're now in the driver's seat: *we* control the scenario or scenarios that a build will execute.

Per Page Assertions/Tests

Even *better*, we have a lot more control *now* over the assertions - or "tests"... Blackfire uses both words - that make a build pass or fail.

For example, the "HTTP requests should be limited to one per page" will be run against *all* pages in the scenarios - that's 2 pages right now. But the homepage *also* has its *own* assert: that the SQL queries on this page should be less than 30. If you look back at the build... we can see that assertion! We can even click into the profile, click on "Assertions", and see both there.

So not *only* do we have a lot of control over *which* pages we want to test - even including filling out forms - but we can *also* do custom assertions on a page-by-page basis in addition to having global tests. I *love* that. And now I can remove the comment I put earlier above assert:

23 lines .blackfire.yaml

```
... lines 1 - 6
scenarios: |
... lines 8 - 9
scenario
... lines 11 - 12
visit url('/')
... lines 14 - 16
# won't work until we're using Blackfire environment
... lines 18 - 23
```

Now that we're running this from inside an environment, this *does* work.

Next, let's use our power to *carefully* add more time-based assertions on a page-by-page basis. We'll also learn how you can add your *own* metrics in order to, well, write performance assertions about pretty much *anything* you can dream up.

Chapter 33: Per-Page Time Metrics & Custom Metrics

We know that the scenario will be executed against our *production* server only. If we profiled a *local* page, this stuff has no effect. That means that the results of these profiles *should* have *less* variability. Not *no* variability: if your production server is under heavy traffic, the profiles might be slower than normal. But, it will have less variability than trying to compare a profile that you created on your local machine with a profile created on production: those are totally different machines and setups.

Tip

I also recommend adding samples 10 to each scenario. This will then use 10 samples (like normal Blackfire profiles) and further reduce variability:

```
visit url("/")
    name "Homepage"
    samples 10
    ...
```

Cautiously Adding Time-Based Assertions

This means that you can... *maybe* add some time-based assertions... as long as you're conservative. For example, on the homepage, let's assert that main.wall_time < 100ms:

23 lines .blackfire.yaml

```
... lines 1 - 6
scenarios: |
... lines 8 - 9
scenario
... lines 11 - 12
visit url('/')
... lines 14 - 17
assert main.wall_time
... lines 19 - 23
```

By the way, *most* metrics start with metrics. and you can look on the timeline to see what's available. A *few* metrics - like wall time and peak memory - start with main..

Anyways, as you can see inside Blackfire, our homepage on production *normally* has a wall time of about 50ms... so 100ms is *fairly* conservative. But time-based metrics are *still* fragile. Doing this *will* likely result in some random failures from time-to-time.

Let's commit this:

```
git status
git add .
git commit -m "adding homepage time assertions"
```

And deploy:

```
symfony deploy --bypass-checks
```

Custom Metrics

While that's deploying, I want to show you a *super* powerful feature that we won't have time to experiment with: custom metrics. Google for "Blackfire metrics". In addition to the timeline, this page *also* lists *all* of the metrics that are available.

But you can also create your *own* metrics inside .blackfire.yaml. In addition to tests and scenarios, we can have a metrics key. For example, this creates a custom metric called "Markdown to HTML". The *real* magic is the matching_calls config: any time the toHtml method of this made-up Markdown class is called, its data will be *grouped* into the markdown_to_html metric.

That's powerful because you can immediately use that metric in your tests. For example, you could assert that this metric is called exactly zero times - as a way to make sure that some caching system is *avoiding* the need for this to ever happen on production. Or, you could check the memory usage... or other dimension.

You can use some pretty serious logic to create these metrics: making it match only a specific *caller* for a function, OR logic, regex matching and ways to match methods, calls from classes that implement an interface and many other things. You can even create *separate* metrics for the *same* method based on which *arguments* are passed to them. They went a little nuts.

Checking the Time-Based Metric

Anyways, let's check on the deploy. Done! Go back - I'll close this tab - and let's create a new build. Call it "With homepage wall time assert". Start build!

And... it passes! This time we can see an extra constraint on the homepage: wall time needs to be less than 100ms. If it's *greater* than 100ms and you have notifications configured, you'll know immediately.

Next: now that we have this idea of builds being created every 6 hours, we can do some cool stuff, like *comparing* a build to the build that happened before it. Heck we can even write *assertions* about this! Want a build to fail if a page is 30% *slower* than the build before it? We can do that.

Chapter 34: Testing a Build Compared to the Last Build

A *long* time ago in this tutorial, we talked about Blackfire's *truly* awesome "comparison" feature. If you profile a page, make a change, then profile it again, you can *compare* those two profiles to see *exactly* how that change impacted performance.

When you use the build system, you can do the *exact* same thing... and you can *even* write "tests" that compare a build to the *previous* build. For example, you could say:

> Yo! If the wall time on the homepage is *suddenly* 30% *slower* than the previous build, I want this build to fail.

Adding a Comparison Test with percent()

How can we do that? It's dead simple. Add a new global metric - how about "Pages are not suddenly much slower" - and set this to run on every page: path: /.*. For the assertion, we can use a special function called percent: percent(main.wall_time) < 30%:

27 lines .blackfire.yaml

```
"tests":
... lines 2 - 5
"Pages are not suddenly *much* slower":
path: "/.*"
assertions:
- "percent(main.wall_time)
... lines 10 - 27
```

That's it! There's also a function called diff(). If you said diff(metrics.sql.queries.count) < 2 it means that the *difference* between the number of SQL queries on the new profile minus the old profile should be less than 2.

Let's see what this looks like! Find your terminal and commit these changes:

```
git status
git add .
git commit -m "adding global wall time diff assert"
```

Now... deploy!

```
symfony deploy --bypass-checks
```

Comparison Tests: Not for Manual Builds

But... bad news. If we waited for that to finish deploying... and then triggered a new custom build... that test would *not* run. In fact, I want you to see that. Wait for the deploy to finish - okay, good - then move back over and start a build.

This does what we expect: it executes our scenario and creates 2 profiles. Look at the 3 successful constraints for the homepage: we see the other global test about "HTTP requests should be limited"... but we don't see the new one. What gives?

So... when you create a build, you can specify a "previous" build that it should be compared to by using an internal "build id". Our project is too new to see it, but this happens automatically with "periodic" builds: our comparison assertion *will* execute on periodic builds.

Tip

Triggering builds via a webhook requires an Enterprise plan.

But when we create a manual build... there's no way to specify a "previous" build... which is why the comparison stuff doesn't work. *Fortunately*, since I don't want to wait 12 hours to see if this is working, there is *another* way to trigger a build: through a webhook. Basically, if you want to create a build from outside the Blackfire UI, you can do that by making a request to a specific URL. And when you do that, you can *optionally* specify the "previous build" that this new build should be compared to.

Automatic Build on Deploy

This webhook-triggered-build is *especially* useful in one specific situation: creating a build each time you *deploy*. If you did that correctly, your comparison assertion would compare the latest deploy to the *previous* deploy... which is pretty awesome.

Because we're using SymfonyCloud, this is dead-simple to set up.

Find the Blackfire SymfonyCloud documentation and, down here under "Builds", I'll select our environment. Basically, by running this command, we can tell SymfonyCloud to send a webhook to create a Blackfire build *each* time we deploy.

Copy it, move over to your terminal and... paste:

```
symfony integration:add --type=webhook --url='https://USER:PASS@blackfire.io/api/v2/builds/env/aaaabbee-abcd-abcd-abcd-c49b32bb8f17/symfonycloud'
```

Hit enter to report all events and enter again to report all states. For the environments - this is asking which *SymfonyCloud* environments should trigger builds. Answer with just master - I'll explain why soon.

And... done! Let's redeploy our app. Oh, but before we do, refresh our builds page. Ok, we have 5 builds right now. Now run:

symfony redeploy --bypass-checks

This should be pretty quick. Then... go refresh the page. Yes! A new build - number 6 - triggered by SymfonyCloud. And it *passes*. Awesome! Let's redeploy again:

symfony redeploy --bypass-checks

When that finishes... there's build 7! But to see the comparison stuff in action, I need to do a *real* deploy so that the next build is tied to a *new* Git sha. I'll do a meaningless change, commit, then deploy:

git commit -m "triggering deploy" --allow-empty
symfony deploy --bypass-checks

Seeing the Compared Builds

Actually, I could have skipped changing *any* files and committed with --allow-empty to create an empty commit. When this finishes... no surprise! We have build 8!

On *this* build, it's super cool: each profile has a "Show Comparison" link to open the "comparison" view of that profile compared to the *same* profile on the build from the *last* deploy - which - if you click "latest successful build" - is build 7.

Back on build 8, click the "Show 4 successful constraints" link. There it is! We can see our "Pages are not suddenly *much* slower" assertion! It's comparing the wall time of this profile to the one from the last build.

Click to open up the profile... and make sure you're on the Assertions tab. I love this: 2 page-specific assertions from the scenario, and 2 global assertions: one using the percent() function.

The "Recommendations" *also* got a bit better: Blackfire automatically has some built-in recommendations using diff: this *recommends* that the new profile should have less than 2 *additional* queries compared to the last build. It *looks* like it failed... but that's just because the *other* part of this recommendation - not making more than 10 total queries - failed.

Next: what about running builds on your staging server so you can catch performance issues before going to production? Or what about executing Blackfire builds on each pull request? We can *totally* do that - with a *second* environment.

Chapter 35: Staging Servers on SymfonyCloud

For your site, you *hopefully* have a staging environment - or maybe *multiple* staging environments where you can deploy new features and test them. What about *those* machines? Should we *also* run Blackfire builds on them?

Why Profile Staging Servers?

At first, that *might* not seem important. After all, if a staging machine is a bit slow, who cares? But thanks to the *assertions* we've been writing, if we executed our Blackfire scenarios on a *staging* machine, we could identify performance failures *before* deploying them to production. And if you have a *really* cool setup, you can even have build results posted automatically to your pull request. OooOOoo.

Separating Staging from Production on Blackfire

Getting Blackfire set up on a staging server *seems* simple enough: just repeat the Blackfire installation process... on a different server! But stop! I don't want you to *quite* do that.

Why? I want your Blackfire production environment to *only* contains builds from your *actual* production servers. I want this to be a *perfect* history and representation of production *only*. If we suddenly start adding builds from a staging server - which maybe has different hardware specs... or is running a buggy new feature - some of those builds will fail... and we'll get extra noise in our notifications.

Instead, I like to create a *second* Blackfire environment and send profiles to *it*. If I have *multiple* staging servers, I make them *all* use this same new environment.

SymfonyCloud Environments

But... before we create that *second* Blackfire environment... I need you to - once again - pretend like Blackfire doesn't exist at *all*... for a few minutes.

Because before we talk about how we *profile* a staging server, we need to *create* a staging server and deploy to it. SymfonyCloud has an *incredible* way to do this. *Unfortunately*, the feature in Symfony cloud that *does* this is called... environments. And it has absolutely *nothing* to do with Blackfire environments.

Here's how it works: in addition to your master branch, which is your production server, SymfonyCloud allows you to *deploy* different git *branches*. Each deploy will get its own unique URL. Each branch deployment is called an "environment". If you run:

```
symfony envs
```

Yep! We currently have *one* environment: master. It's the "current" environment because we're checked out to the master git branch locally.

Ok, pretend that we're working on a new feature. And so, we want to create a new local branch for it. Instead of doing that manually, run:

```
symfony env:create some_feature
```

This does two things. First, it created a new local branch called some_feature. That's no big deal: we could have done that by hand. Second, it *deploys* that branch! It does this by creating a "clone" of the master environment: - even creating a copy of the production database!

I'll fast-forward through the deploy. When it finishes, it gives us a URL to the deploy. This is a *different* URL than on production: it's a totally separate, isolated deployment. Let's open this the lazy way:

```
symfony open:remote
```

Say hello to our *staging* server for the some_feature branch, which you can see contains a *copy* of the production database! How cool is that?

Configuring Blackfire on the Environments

Back on Blackfire, refresh to see the builds for the production environment. When we deployed to that environment, it did *not* create a new build. We expected that. When we added the integration to SymfonyCloud - we told it to trigger a build on this Blackfire environment whenever we deploy the *master* branch *only*. We did that because we *don't* want these staging servers to create builds here.

Next, let's create a *second* environment and configure our staging servers to use *it*.

Chapter 36: Staging Environment Builds

We now have *two* versions of our site deployed: our production deploy and a, sort of, "staging" deploy of a pretend feature we're working on. Blackfire is *all* set up on the *production* server, but not on the staging server. Let's fix that!

Back on the "Install" page, select "SymfonyCloud" as our host to get to its docs. To set up Blackfire on production, we did 3 things. One, added the extension. Two, ran this var:set command to configure our Blackfire Server id and token. And three, ran integration:add so that every deploy to master would trigger a Blackfire build in our environment.

Technically, on the staging server, the Blackfire extension is already enabled *and* it's set up to use the Server Id and token from our *production* Blackfire environment. But, as we talked about in the last chapter, I don't want to mix my production builds with builds from staging servers.

Creating a new Blackfire Environment

Instead, go back to our Blackfire organization and create a *second* environment. Let's call it "Sasquatch Sightings Non-master". For the endpoint, use the production environment URL. But don't worry, that URL won't *actually* be used. You'll see.

Hit "Create environment"... then remove the build notifications and save. View the new environment - I'll get the credentials in a minute. Now, *stop* the periodic builds. Why? Well in our setup, at any point, we may have zero or *many* different "staging" servers. There's not just *one* server to build... so if we did a periodic build... which "staging" server would it use? It just doesn't make sense in our case. What *does* make sense is to *trigger* a new build each time we *deploy* to a staging server.

Different Server Id and Token on Staging

Ok, let's think about this: we now have *two* Blackfire environments. We want the *production* server to use the Blackfire server id and token for the production environment... and we want every *other* deploy to use the Blackfire id and token from the new "Non-master" environment.

How you do that depends on how you deploy. For us, we can use a SymfonyCloud config trick. First, list which variables we have set with:

```
symfony vars
```

We have the two that were set by the var:set command we ran earlier. Delete *both* of them:

```
symfony var:delete BLACKFIRE_SERVER_ID BLACKFIRE_SERVER_TOKEN
```

We're going to *re-add* these in a minute... but with some different options. Now, go back to the installation page... and refresh... so this shows our new environment. For the var:set command, select the Non-master environment. Copy the command, move over and paste:

```
symfony var:set BLACKFIRE_SERVER_ID=XXXXXXX BLACKFIRE_SERVER_TOKEN=XXXXXX
```

If we stopped now, it would mean that *every* server would send its profiles to the new Non-Master environment... which is not exactly what we want. But here's the trick: on the "Install" page, change to the "Production" Blackfire environment, and copy *its* command. We're going to *override* these variables, but *just* on the SymfonyCloud master environment.

Paste the command, then add --env=master --env-level so that the variables are used as "overrides" for *only* that environment. Finish with --inheritable=false so that when we create *new* SymfonyCloud environments, they don't inherit these variables from master: we want them to use the *original* values:

```
symfony var:set BLACKFIRE_SERVER_ID=XXXXXXX BLACKFIRE_SERVER_TOKEN=XXXXXX \
    --env=master --env-level --inheritable=false
```

This is a *long* way of saying that the master environment on SymfonyCloud will now use the server id and token for the "Sasquatch Sightings Production" Blackfire environment. And every *other* deploy will use the credentials for the "Non-Master" environment. To be sure, run:

```
symfony vars --env=master
```

Yep! 6900 is the server id for Production. Now try:

```
symfony vars --env=some_feature
```

Perfect: that uses the *other* Server id and token. We're good!

Staging: Builds on Deploy

The *last* thing I want to do is run this integration:add command again. We ran this *earlier* to tell SymfonyCloud that it should notify our "Production" Blackfire environment whenever we deploy to master. Now copy the "Non-Master" environment command... and run it:

```
symfony integration:add --type=webhook --url='https://USER:PASS@blackfire.io/api/v2/builds/env/aaaabbee-abcd-abcd-abcd-c49b32bb8f17/symfonycloud'
```

Say yes to all events, all states *and* all environments. Actually, what we *really* want to say is: create a build on the "Non-Master" environment every time any branch *except* for master is deployed... but I don't think that's possible.

Phew! Let's redeploy both SymfonyCloud environments to see all of this in action:

```
symfony redeploy --bypass-checks
```

Because we're currently checked out to the some_feature branch, this *deploys that* branch. When it finishes, run the same command but with --env=master to redeploy production:

symfony redeploy --bypass-checks --env=master

We also could have *switched* to that branch - git checkout master - and *then* ran symfony redeploy. That's the more traditional way.

Done! Let's go see what that did! First check out the Blackfire production environment. Yes! The redeploy to master created *one* new build. Perfect. Now check out the Non-master environment. Oh, this has *two* new builds: one for the some_feature deploy and another for the master deploy. We don't *really* want or care about that second one... but it's fine. What we *do* care about is that *now*, every time we deploy to a non-production server, we get a new build here.

If you use GitHub or Gitlab, you can take this one step further by doing 2 things. First, SymfonyCloud has a feature where it can automatically deploy the code you have on a pull request. And because that would trigger a new build, *second*, you can configure Blackfire to *notify* GitHub or Gitlab of your build results so that they show up *on* the pull request itself. Pretty awesome.

I *love* our setup. But there's one more environment feature that we haven't checked out yet: the ability to set *variables* that you use in your scenarios. Let's check that out next.

Chapter 37: Blackfire Environment Variables

Often, your production server will have different - hopefully *bigger* - hardware than your staging server... which means that your staging builds may run slower than production. That's going to be a problem if you have *time* based metrics: the wall time of a build may be less than 100ms on production... but *more* than that on staging:

27 lines .blackfire.yaml

```
... lines 1 - 10
scenarios: |
... lines 12 - 13
scenario
... lines 15 - 16
visit url('/')
... lines 18 - 21
assert main.wall_time
... lines 23 - 27
```

That means the staging builds will always fail. Bummer!

Hello Build Variables

No worries. To help, each environment can define *variables*. Check it out: inside the metric expression, I'll add a set of parentheses around the 100ms and then say *times* and call a var() function. I'll invent a new variable: speed_coefficient and give it a *default* value - via the 2nd argument - of 1:

27 lines .blackfire.yaml

```
... lines 1 - 10
scenarios: |
... lines 12 - 13
scenario
... lines 15 - 16
visit url('/')
... lines 18 - 21
assert main.wall_time
... lines 23 - 27
```

Now, when this assertion is executed, it will assert that the wall time is less than 100ms *times* whatever this speed_coefficient variable is. What *is* speed_coefficient? It's *totally* something I just made up and it is *not* set anywhere. Where *do* we set it? Inside our Blackfire environment!

Copy the variable name and go into the Non-Master environment. On the right, near the bottom, click the pencil icon to edit our variables. Add the variable set to... how about 2: that will allow the staging server to be *twice* as slow.

Do we *also* need to set this inside the "Production" environment? Nope: I'll just let it use the default value of 1.

Let's try it! Spin back over to your terminal, add the change... and commit:

```
git add .
git commit -m "adding speed_coeffient variable for wall time assert"
```

As a reminder, we're on the some_feature branch. So when we run:

```
symfony deploy --bypass-checks
```

We're deploying to *that* environment.

Seeing the Variable in Action

When that finishes... move back over to the Blackfire environment, refresh and... hello new build! Look inside. There are two cool things. First, under the homepage, you can see the speed_coefficient in action - the little "2" tells us the value it's using. So, in reality, it's asserting that 50.8ms is less than *200* milliseconds.

Feature Branch Comparisons

The *other* thing I want you to notice is that, if you go back to the builds page, we have now built the some_feature branch *twice*. When you click on the second, newer build, it has the *comparison* stuff! It allows us to *compare* this build to the *previous* commit on the *same* branch. This allows you to see - commit-by-commit - *when* a feature started having performance problems.

And... that's it for the Blackfire tutorial! I hope you *loved* this nerdy trip into the *depths* of performance as much as I did. Blackfire can give you a *lot* of info immediately... or you can *really* dive in and make it *sing*. Personally, I love having the builds and this performance history for SymfonyCasts.com. Oh, and

a special thanks to Jérôme Vieilledent - I almost *definitely* just slaughtered his name - for his endless patience answering my hundreds of Blackfire questions.

And as always, if *you* have any questions... or we didn't explain something you wanted to know about... or you want a cake recipe... we're here for you in the comments. If you have any *serious* performance wins, we would *love* to hear about them.

Alright friends - I wish you a *speedy* day! Seeya next time!