Question and Answer Day: March 27th, 2013

With <3 from KnpUniversity

Chapter 1: Ask Questions!

ASK QUESTIONS!

Hi guys! It's time for KnpU Question and Answer day: your chance to submit your burning programming questions and our chance to listen for a change! Here's how it works:

1) **Ask your questions** about PHP, Symfony, Behat, dinosaurs, or anything else as as a *comment* on this page before the closing time and we'll pretend to know the answers.

Note

You have until the end of Tuesday, March 26th, 2013 to post questions!

(Psssst - the comments are over on the right side ==========>)

2. The next day, we'll open this page up and start answering them! Follow us on Twitter and if you include your Twitter handle with your question, we'll give you a shout-out.

Note

Answer day is Wednesday, March 27th, 2013. Follow us on twitter@KnpUniversity

That's it! We really want to hear what questions you're having in your real projects. We can't guarantee we'll be able to answer every question, but we'll give it the old college try ;).

And remember any question or code gists you post are public. So, it may be a good idea to not tell the world what your database password is :)

See ya then!

Chapter 2: More on Routing And Dependency Injection Parameters

MORE ON ROUTING AND DEPENDENCY INJECTION PARAMETERS

From Dimka Mo

> How can I hide the pattern for a route - e.g. via parameter in `parameters.yml`? My goal looks like: i have some line in `parameters.yml`:

```
secret_url: /are/you/a/robot/
```

then in `routing.yml` something like this:

```
pattern: %secret_url%
defaults: { ... }
```

I need to hide my url pattern from public, how can i do this? Thanks!

Answer

I don't think you were giving yourself enough credit with your question, because you already know the answer! ;).

As we mentioned in the Hostname Routing chapter of our What's new in Symfony 2.2 tutorial, starting in Symfony 2.1, you can use a dependency injection parameter anywhere in your routing.

First, let's start with a normal route:

```
# app/config/routing.yml
# ...

parameter_test:
    path: /are/you/a/robot
    defaults: { _controller: QADayBundle:Default:parameterTest }
```

The goal is to move the `/are/you/a/robot` part out of our code to somewhere that's not committed. For many of you, that may be a strange requirement, but the exercise here highlights a lot of nice things.

Tip

And remember, there's nothing special about the `parameters.yml` file, except that we *choose* to put server-specific code in that file because we don't commit it to the repository (if this is new to you, see Getting Started in Symfony2).

First, add a new parameter to your `parameters.yml` file:

```
# app/config/parameters.yml
# ...

my_hidden_url: /are/you/a/robot
```

To finish this off, simply reference it in your route:

```
# app/config/routing.yml
# ...

parameter_test:
    path: "%my_hidden_url%"
    defaults: { _controller: QADayBundle:Default:parameterTest }
```

That's it! But let's see what else we can do!

Using a Parameter as *part* of the routing Path

You can also leverage parameters as just a part of your routing path. To show this off, create a new route to play with:

```
# app/config/routing.yml
parameter_prefix:
    path: /admin/test
    defaults: { _controller: QADayBundle:Default:parameterTest }
```

If you had a lot of routes that began with the `/admin` prefix, you might not want to repeat yourself. One solution of course is to import these routes from an external routing file and use the prefix key.

But you can also use parameters. This time, let's add a new parameter directly to our `config.yml` file. I'm deciding to put it here instead of inside `parameters.yml` because this value isn't secret or server-specific:

```
# app/config/config.yml
parameters:
    admin_prefix: /admin
```

We can now use this just like before, but now forming just a part of our routing path:

```
# app/config/routing.yml
parameter_prefix:
    path: "%routing_prefix%/test"
    defaults: { _controller: QADayBundle:Default:parameterTest }
```

Extra Credit: Where does this Magic Happen?

Dependency injection parameters like `%routing_prefix%` are part of building Symfony's service container: you define services and parameters, and when the whole container is built, any strings surrounded by two `%` signs are replaced by that parameter value.

But the engine that builds the service container is completely different from the engine that compiles your routes together. So where do the two cross over?

The answer is in the `Router` class that's used inside the Symfony Framework. Symfony's Routing Component supplies a Router class which handles matching and generating URLs. But when you use the Symfony Framework, the actual Router object you're using lives in the FrameworkBundle. In fact, this is really common, and you can see the class of these objects by finding the service via the `container:debug` command:

```
php app/console container:debug | grep -i router
```

```
router container Symfony\Bundle\FrameworkBundle\Routing\Router
```

If you scan the list, the `router` service should jump at you. Indeed, the "router" used in the Symfony Framework is an instance of Symfony\Bundle\FrameworkBundle\Routing\Router.

The routing parameter magic happens in `getRouteCollection`:

```
public function getRouteCollection()
{
    if (null === $this->collection) {
        $this->collection = $this->container
            ->get('routing.loader')
            ->load(
                $this->resource,
                $this->options['resource_type']
            );
        $this->resolveParameters($this->collection);
    }

    return $this->collection;
}
```

This method is called early on when Symfony needs the full collection of routes to use. The key here is that before returning the collection, the resolveParameters function is called, which iterates over every route in the collection and replaces parameters in the `defaults`, `path`, `requirements` and `host` keys of the route.

Why isn't this Slow?

If you're wondering if iterating over every single route to replace this parameter is slow, the answer is YES! But in reality, not at all :). In the Symfony2 Framework, the final collection of routes is dumped to a physical file that lives in your cache directory. It means that this process happens once, then never again until your cache needs to be rebuilt.

Modifying Routes On-the-fly

You should never be in a hurry to extend Symfony and add a lot of magic to it, but this is a great example of a way that you can do just that. Imagine that there was some modification that you needed to make to every single route in your system that couldn't be accomplished by leveraging a parameter. One way to accomplish this would be to sub-class the `Router` class, override `getRouteCollection`, and make your own changes.

... but for now I'll leave that as an exercise for you :).

Chapter 3: How to use Behat and Selenium on Travis CI

HOW TO USE BEHAT AND SELENIUM ON TRAVIS CI

From spolischook

> How to connect Symfony2 project with Behat and Sahi to Travis Ci

Answer

Note

Special thanks to our very-own Roman on this answer!

GREAT question, and one we've struggled and dealt with quite a bit over the last few months. Fortunately, we have it working now - it's not always perfect, but this should get your started.

Our goal will actually be to configure our `.travis.yml` file to execute our Behat tests, some of which require Selenium. We like Selenium over Sahi because it's very well-supported and generally seems to run just a bit faster.

I'll assume that you already have Behat installed with a few `@javascript` features you'd like to run and focus specifically on the `.travis.yml` configuration. And if you're looking to sharpen your Behat skills (or start using it!), check out BDD, Behat, Mink and other Wonderful Things.

1) Installing a Web Server (e.g. Apache)

Your Travis server is an open canvas, upon which we computer-science artist may paint whatever software and configuration we want. So... let's start with a web server!

```
before_script:
    - sudo apt-get update > /dev/null
    - sudo apt-get install -y --force-yes apache2 libapache2-mod-php5 php5-curl php5-mysql php5-intl
```

Note

Don't forget to install all other non-default extensions (i.e. php-ssl)

2) Give yourself a VirtualHost

Since we'll be making real HTTP requests back to our application, we'll need a VirtualHost setup. One easy way to do this is to leverage Apache's `default` VirtualHost, and use `sed` to stretch it to our needs:

```
before_script:
    # ...
    - sudo sed -i -e "s,/var/www,$(pwd)/web,g" /etc/apache2/sites-available/default
    - sudo /etc/init.d/apache2 restart
```

And if you want to use a specific domain, you can set that up too: just be sure to do it before the apache restart call:

```
before_script:
    # ...
    - sudo sed -i -e "/DocumentRoot/i\ServerName knpu_qa.l" /etc/apache2/sites-available/default
    - echo "127.0.0.1 knpu_qa.l" | sudo tee -a /etc/hosts
    - sudo /etc/init.d/apache2 restart
```

3) Composer! And all the other Stuff

Since Travis takes care of pulling your project into the server at the right version, we just need to download any dependencies we have. We're using Composer and if you're not, you'll just need to tweak these commands to download your libraries dependencies, however that may be:

Tip

The `composer` executable is available on your Travis machine by default, but it may not be the latest version.

```
before_script:
    # ...

    # it may be useful to have the latest composer
    - composer self-update
    - composer install --dev --prefer-dist
```

The `--prefer-dist` part of Composer tells it to try to download zip archives, instead of cloning the repositories of your dependencies. We've chosen to do this because it's a lot faster. However, we've found if your packages are hosted on GitHub, then you may see intermittent failures downloading the packages. There's not much you can do here, but you may try `--prefer-source`, which will be slower, but *potentially* more reliable.

4) App-specific Stuff

We now have a web server, a virtual host, our application and its dependencies all ready to go. Now it's your turn to initialize the database, set any file permissions, and anything else you may need to do before your application is fully functional.

For Symfony2, the following code should do the trick (or at least get you started):

```
before_script:
    # ...

    - app/console do:da:cr -e=test > /dev/null
    - app/console do:sc:cr -e=test > /dev/null
    - chmod -R 777 app/cache app/logs
    - app/console --env=test cache:warmup
    - chmod -R 777 app/cache app/logs
```

Note

Yes, the double `- chmod -R 777 app/cache app/logs` is on purpose. Because multiple users will touch the cache files, we've had the most success warming all of the files and then once again making sure they're all writable.

5) The Selenium Magic

And finally, the step you've been waiting for: how the heck do I run Selenium in this windowless machine? One solution that we've had success with is by leverage a utility called `xvfb`, or "X virtual framebuffer". It's actually exactly what we want: it does everything that X does... but without there actually being a window. Cool!

So let's get it all installed:

```
before_script:
    # ...

    - "sh -e /etc/init.d/xvfb start"
    - "export DISPLAY=:99.0"
    - "wget http://selenium.googlecode.com/files/selenium-server-standalone-2.31.0.jar"
    - "java -jar selenium-server-standalone-2.31.0.jar > /dev/null &"
    - sleep 5
```

The reason we need `sleep 5` at the end is because the selenium server takes just a bit of time to initialize. If it's not ready when Behat starts, then all related tests will fail for this build. Eek!

If you're curious about any more of this, check out the GUI & Headless browser testing on travis-ci.org by the Travis folks.

Tip

You might want to use Chrome instead of the default (Firefox), since it's a bit faster and more stable. If so, try this:

```
- "wget http://chromedriver.googlecode.com/files/chromedriver_linux32_23.0.1240.0.zip && unzip chromedriver_linux32_23.0.1240.0.zip && sudo mv chromedriver /usr/bin"
```

6) Running your tests

Ok, let's do this! To run your tests... just run your tests! For example, suppose we have some PHPUnit tests along with our Behat tests:

```
script:
    - phpunit path/to/tests
    - bin/behat
```

For Symfony2, this will look a bit different:

```
script:
    - phpunit -c app src/
    - bin/behat @KnpQABundle
    - bin/behat @KnpAnotherBundle
```

7) Other Issues and Improvements?

I'll be honest, it's tough to get this stuff right, especially since you can't shell directly to the server and look around. Phantom GitHub download failures may also cause some heartache.

Have you found some other tricks and secrets you want to share? Do it!

Here are a few other complications you may encounter:

GitHub API Rate Limit

If you have a lot of dependencies, you may eventually see this awesome error in your Travis output:

> Could not fetch https://api.github.com/repos/Behat/MinkGoutteDriver/zipball/v1.0.7, enter your GitHub credentials to go over the API rate limit

No worries! To fix this, you can use your own account to get a token that your Travis build can use to get around this. We have this working here at KnpUniversity.com, and we stole the whole idea from this blog: Creating and Using a Github OAuth Token With Travis And Composer.

The end-result is a `.travis.composer.config.json` file that looks like this:

```
{
  "config":{
    "github-oauth":{
      "github.com":"5675git-yer-own-key9854abc"
    }
  }
}
```

and a new entry in `.travis.yml` before updating your composer dependencies:

```
before_script:
    # ...

    - "mkdir -p ~/.composer"
    - cp .travis.composer.config.json ~/.composer/config.json
```

8) Celebrate!

That's it! Crack open an ice-cold beer, spiced vanilla latte, cold water, goat's milk, or whatever your preferred beverage and watch as Travis does all the work for you.

But seriously, if you have any issues or improvements, post them for everyone! Travis is still somewhat new, so it’s a living process.

Cheers!

Chapter 4: Creating your very own Composer Package

CREATING YOUR VERY OWN COMPOSER PACKAGE

From: Marcin Grochulski

> I wonder how to create your own bundle and then add it as an installation package for the Composer.

Answer

This is a *great* Composer question, and will let us walk through the lifecycle of a library and how it works with Composer. Be sure to check out our free Wonderful World of Composer screencast first before diving in here.

Let's suppose that we have a library or Symfony2 Bundle, and we'd like to release this open source and then include it in our projects. You can do this at a number of different levels of sophistication. Let's walk through it!

Step 1: Put your Library on GitHub

Before anything else, put your library on GitHub. Seriously, if you *only* did this, then people could already begin using your library.

In fact, I've just created a wonderful new library that does... well, nothing honestly - but it'll serve as our example: https://github.com/weaverryan/derp-dangerzone.

The library is up on GitHub, and in real life would actually have some useful things. You'll also see a `composer.json` file. ignore it and pretend it isn't there for now.

Now suppose that we want to include that library in one of our projects. If the new library were registered with Packagist (we'll add it eventually), then it would be as simple as adding one line to our `require` key in `composer.json`.

But since it's not, we have to do the work ourself using a custom repositories key in the `composer.json` or our *project*:

```
"repositories": [
    {
        "type": "package",
        "package": {
            "name": "weaverryan/derp-dangerzone",
            "version": "dev-master",
            "source": {
                "url": "git://github.com/weaverryan/derp-dangerzone.git",
                "type": "git",
                "reference": "master"
            },
            "autoload": {
                "psr-0" : {
                    "Weaverryan\\DangerZone" : "src"
                }
            }
        }
    }
],
```

Tip

The `repositories` key sits at the root of your `composer.json` file, as a sibling to (i.e. next to) the `require` key.

Wow, that was a lot of work! The problem is that the `derp-dangerzone` doesn't have a `composer.json` file yet (well, we're pretending it doesn't), so we have to manually define the package ourselves. There are a few interesting parts:

- `version`: Our library doesn't really have versions yet, so we create a single version that points to the

`master` branch (see the `reference` key). If we had a real version, we might define something like `2.0.0` here and update the `reference` below to point at a branch or tag.
- `autoload`: Most libraries follow the PSR-0 naming standard, including our new library. The only class in the library is in the `Weaverryan\DangerZone` namespace and is called `HalloThere`. Accordingly, once you're in the `src/` directory, it lives at `Weaverryan/DangerZone/HalloThere.php`. Under this key, we tell Composer that all of our classes will live in the `Weaverryan\Dangerzone` namespace and to start looking for them in the `src/` directory.

With this new entry, Composer now sees a fully valid package called `weaverryan/derp-dangerzone` with a single `dev-master` version. In other words, just add it to your `require` key in the `composer.json` of your *project*:

```
"require": {
   "... other libraries": "... other version",

   "weaverryan/derp-dangerzone": "dev-master"
},
```

Update as you normally do:

```
php composer.phar update
```

Phew! That was a lot of work. But as we make our library more official, most of the work is behind us!

Step 2: Give your Library a composer.json File

If everyone that uses your library needs to do all that work, you can bet that you won't be very popular. To fix this, we'll need to put a `composer.json` file in the library itself. Fortunately, this is really easy, and we can basically move the `package` we already created into a new `composer.json` file at the root of our library. To make it easier, you can remove the `version` and `source` keys - Composer will look at your branches and tags to get all of this.

In other words, create a `composer.json` file in your *library*:

```
{
   "name": "weaverryan/derp-dangerzone",
   "autoload": {
      "psr-0" : {
         "Weaverryan\\DangerZone" : "src"
      }
   }
}
```

And this is exactly what you see right now at weaverryan/derp-dangerzone. At this point, the Packagist repository doesn't know about our library, but our library *does* have a `composer.json` file. This is a *huge* step forward, because it lets us simplify our *project's* composer.json quite a bit. We still need a custom `repositories` key, but now it's much simpler.

Update your *projects's* `composer.json` to have the following:

```
"repositories": [
   {
      "type": "vcs",
      "url": "https://github.com/weaverryan/derp-dangerzone"
   }
],
```

Now, instead of a `packages` key, we have a simpler vcs key, which basically says: "go over to this repository and consume its `composer.json` file".

Step 3: Registering with Packagist

As we've seen, creating a `composer.json` file in your library is optional, but makes using it much much easier. The next and last step to simplicity is to register it with Packagist. This is the easiest step yet and involves filling in a few forms at Packagist and waiting for it to crawl your repository.

Once you've registered your library with Packagist (and it's been crawled), your library can be used by adding a single entry to the `require` key of a `composer.json` file: no extra `repositories` entry is needed:

```
"require": {
    "... other libraries": "... other version",

    "weaverryan/derp-dangerzone": "dev-master"
},
```

That's it! The process is simple, but nice to walk through. Now start sharing your code!

Chapter 5: Swiftmailer Spooling and Handling Failures

SWIFTMAILER SPOOLING AND HANDLING FAILURES

From Philipp Rieber

> Hi, I'm using swiftmailer's file spooling and I'm flushing the queue every minute using a cron task:
>
> ```
> app/console swiftmailer:spool:send --env=prod > /dev/null 2>>app/logs/error.log
> ```
>
> Due to SMTP errors like "554 Message rejected: Address blacklisted" or "554 Message rejected: Email address is not verified" some message files remain in the spool directory and swiftmailer tries to send them over and over again following the "recovery-timeout" setting of the command (default = 15 minutes).
>
> The problem is that a single exception during the sending process cancels the whole command. So if there are more than 15 "xxx.message.sending" files in the spool directory after a while and the cron job runs every minute with a recovery-timeout of 15 minutes, then the new messages won't get sent any more. How can I handle that? Do I need an additional command to remove old "xxx.message.sending" files, e.g by wrapping and extending the `swiftmailer:spool:send` command?
>
> Currently I remove the old files manually from time to time and according to Google I'm the only one having this issue ;-)
>
> Thank you!

Answer

Woh, tough question! So, let's see what we can do. First, let's me give everyone else a little background by building a test project. Even if you're not having this issue, we're going to learn quite a bit about spooling and some lower-level parts of Swift Mailer. Philipp, you can skip down to the answer, or suggested approach for this difficult problem ;).

First, configure Swift Mailer to send emails in some way, and tell it to use a "file" spool. If you haven't seen this before, we have a cookbook article on it at Symfony.com called, well, How to Spool Emails:

```
# app/config/config.yml
swiftmailer:
    transport: %mailer_transport%
    host:      %mailer_host%
    username:  %mailer_user%
    password:  %mailer_password%
    spool:     { type: file }
```

By default, most of the `swiftmailer` configuration is stored in the `app/config/parameters.yml` file, so make sure you update your settings there.

File spooling is really easy, and kinda neat. Whenever you tell Swiftmailer to send an email, it actually doesn't. Instead it stores it in a file and waits for you to run a Symfony task that actually sends the email. The obvious advantage is that the experience for your end-user is much faster.

Let's use a small script I've created that loads up a bunch of spooled messages for us. This bootstraps Symfony and lets me write any Symfony code I want in it. It's a quick and dirty way to create a spot where we can execute some code that needs Symfony and is something we cover in our Starting in Symfony2 series:

```php
<?php
// load_emails.php
require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
$loader = require_once __DIR__.'/app/bootstrap.php.cache';
require_once __DIR__.'/app/AppKernel.php';
$kernel = new AppKernel('prod', true);
$request = Request::createFromGlobals();
$kernel->boot();
$container = $kernel->getContainer();
$container->enterScope('request');
$container->set('request', $request);
/* end bootstrap */

/** @var $mailer \Swift_Mailer */
$mailer = $container->get('mailer');

$message = \Swift_Message::newInstance()
    ->setSubject('Testing Spooling!')
    ->setFrom('hello@knpuniversity.com')
    ->setTo('ryan@knplabs.com')
    ->setBody('Hallo emails!')
;

for ($i = 0; $i < 10; $i++) {
    $mailer->send($message);
}
```

The script sends 10 email messages. Behind the scenes, I'll also add a little bit of code to the core os Swift Mailer so that my SMTP server appears to fail about every 5 sends. This will fake STMP sending errors:

```php
// vendor/swiftmailer/swiftmailer/lib/classes/Swift/Transport/AbstractSmtpTransport.php
// ...

protected function _assertResponseCode($response, $wanted)
{
    list($code) = sscanf($response, '%3d');

    if (rand(1, 5) == 5 && in_array(250, $wanted)) {
        $code = 554;
    }

    // ... the rest of the function
}
```

How Emails are File Spooled

Run this script from the command line to queue the 10 messages:

```
php load_emails.php
```

Tip

The script runs in the `prod` environment to be more realistic (since your site typically runs in the `prod` environment). So, be sure to clear your `prod` cache before trying any of this:

```
php app/console cache:clear --env=prod
```

You won't see anything visually, and no emails were sent, but if you look in the cache directory, you should see a `swiftmailer` directory with a single file for each spooled message:

```
ls -la app/cache/prod/swiftmailer/spool
```

```
0Mo4LSRwTj.message
30MJF9qOP7.message
BLxbfA_cKs.message
BaW2_ZzpAE.message
CgyPxTQ59E.message
Fw_Bux5LUh.message
GsDgqNHc89.message
IDbFa9CCtB.message
LEw9Xe.EZY.message
RKbbDMVKu9.message
```

This is how the file spool works: each message is given a random filename and its contents are a serialized version of the `Swift_Message`.

To actually send these emails, use the `swiftmailer:spool:send` command.

```
php app/console swiftmailer:spool:send --env=prod --message-limit=10
```

Under normal conditions, this would find the first 10 files in the `spool` directory, unserialize each file's contents and then send it. In fact, behind the scenes, each file is suffixed with `.sending` the moment before it is sent, and then deleted afterwards if everything went ok. If you watched your `spool` directory closely, you could see this while it's sending:

```
0Mo4LSRwTj.message.sending
30MJF9qOP7.message
BLxbfA_cKs.message
BaW2_ZzpAE.message
CgyPxTQ59E.message
Fw_Bux5LUh.message
GsDgqNHc89.message
IDbFa9CCtB.message
LEw9Xe.EZY.message
RKbbDMVKu9.message
```

Normally you don't really care about this... until your emails start to fail.

How Swift Mailer handles Failures

As Philipp mentioned, when you run the `swiftmailer:spool:send` command and one email fails, it will blow up! That's actually not that big of a problem initially: as soon as any email is sent successfully, its spool file is deleted, which avoids duplicate sending, even if another email send blows up later. The email that failed remains in its "sending" state, meaning it has the `.sending` suffix:

```
0Mo4LSRwTj.message.sending
```

When you re-run the command, that `.sending` file is skipped, and the other nine files in the spool are sent.

So then, what happens to the email that failed? Does Swift Mailer every try to send it again? In fact, it does! And this is where the problems start. When you run the command, there is an optional `--recover-timeout` option, which defaults to 900, or 15 minutes. This option means that if a file has been in the `.sending` state for 15 minutes, the suffix should be removed and we should try re-sending it. This is really smart, because it means that if your SMTP server has a temporary failure, the email will just send later.

Failures, Failures Blocking Everything!

But sometimes, an email fails to send for a permanent reason, like `554 Message rejected: Address blacklisted`. No matter how many times you try to re-send that email, it will probably never work. It will fail, wait fifteen minutes, fail again, then repeat endlessly. Even if these happen every now and then, after awhile you'll get a `spool/` directory that's full of failures:

```
0Mo4LSRwTj.message.sending
30MJF9qOP7.message.sending
BLxbfA_cKs.message.sending
BaW2_ZzpAE.message.sending
CgyPxTQ59E.message.sending
Fw_Bux5LUh.message.sending
GsDgqNHc89.message.sending
IDbFa9CCtB.message.sending
LEw9Xe.EZY.message.sending
RKbbDMVKu9.message.sending
```

These are just annoying at first, since after fifteen minutes, each is re-tried, which causes your script to fail and no other emails to be sent. If you're running the script often enough, it's no big deal.

So back to Philipp's question:

> So if there are more than 15 "xxx.message.sending" files in the spool directory after a while and the cron job runs every minute with a recovery-timeout of 15 minutes, then the new messages won't get sent any more. How can I handle that?

Let's walk through this: imagine you have 15 files that are failing. One-by-one, these become eligible to be re-tried. Our script, which runs every minute, tries one, then fails. A minute later it tries another, then another, etc, etc. After fifteen minutes it hasn't actually sent any emails - it's only failed to re-send these. To make matters worse, the first failed email is ready to be re-tried again, so the cycle continues.

The Solution?

This is actually a really interesting, but challenging issue. At the core is the fact that Swift Mailer can't tell the difference between a mail that should be re-tried, and one that will fail forever. To make matters worse, there's no possible way to configure the file spool to stop trying after a few attempts and delete the mail. This seems like a shortcoming in the spool itself, but for now, let's work around it!

In my opinion, the best solution is create a separate task that handles these failures by trying them once more, then deleting them finally. Let's start with the skeleton for the command:

```
namespace KnpU\QADayBundle\Command;

use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class ClearFailedSpoolCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setName('swiftmailer:spool:clear-failures')
            ->setDescription('Clears failures from the spool')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
    }
}
```

The goal of the command will be to find all `.loading` files, try them once again, then delete the spool file. This will use a few parts of Swift Mailer and its integration with Symfony that are deep enough that you'll need to be more careful when you upgrade. For example, the fact that the failed spools are suffixed with `.sending` is really a detail that we're not supposed to care about, but we'll take advantage of it.

To start, grab the *real* transport from the service container and make sure it's started:

```
/** @var $transport \Swift_Transport */
$transport = $this->getContainer()->get('swiftmailer.transport.real');
if (!$transport->isStarted()) {
    $transport->start();
}
```

The “transport” used by the `mailer` service is the file spool, which means when you send through it, it actually just spools. Symfony stores your *real* transport - whether that be SMTP or something else - as a service called `swiftmailer.transport.real`.

Next, let’s find all the spooled files. This takes advantage of the `swiftmailer.spool.file.path` parameter, which contains the directory where the spool files live. This parameter is used when the Swift_FileSpool is instantiated. We’ll also use the Finder component to really make this shine:

```
// ...
$spoolPath = $this->getContainer()->getParameter('swiftmailer.spool.file.path');
$finder = Finder::create()->in($spoolPath)->name('*.sending');

foreach ($finder as $failedFile) {
    // ...
}
```

Finally, fill in the loop:

```
// ...
foreach ($finder as $failedFile) {
    // rename the file, so no other process tries to find it
    $tmpFilename = $failedFile.'.finalretry';
    rename($failedFile, $tmpFilename);

    /** @var $message \Swift_Message */
    $message = unserialize(file_get_contents($tmpFilename));
    $output->writeln(sprintf(
        'Retrying <info>%s</info> to <info>%s</info>',
        $message->getSubject(),
        implode(', ', array_keys($message->getTo()))
    ));

    try {
        $transport->send($message);
        $output->writeln('Sent!');
    } catch (\Swift_TransportException $e) {
        $output->writeln('<error>Send failed - deleting spooled message</error>');
    }

    // delete the file, either because it sent, or because it failed
    unlink($tmpFilename);
}
```

Woh! Let’s walk through this using 4 friendly bullet points:

1) We rename the spool file to prevent any other process from sending this file while we try;

2) The contents of the spool file are a serialized `\Swift_Message` object, which we an unserialize to get it back;

3. We once again try to `send` the message.

4) Whether the message sends or fails, we delete the spool file to clean it out.

And that’s it! Now, set the command to run on some interval. If these messages tend to start to be a problem after an hour, run this hourly. If it’s an uncommon issue, run it daily:

```
php app/console swiftmailer:spool:clear-failures --env=prod
```

With a good mixture of failures and success, the output will look something like this:

```
Retrying Testing Spooling! to ryan@knplabs.com
Sent!
Retrying Testing Spooling! to ryan@knplabs.com
Send failed - deleting spooled message
Retrying Testing Spooling! to ryan@knplabs.com
Sent!
Retrying Testing Spooling! to ryan@knplabs.com
Send failed - deleting spooled message
```

There are countless other approaches you could take, but I prefer this one because it prevents you from needing to override any core code. The point is that, one way or another, you’re on your own when you solve

this. With some refactoring of `Swift_FileSpool`, it should be possible to set a max retry limit per mail, but that's not the case right now.

Still, file spooling is great. If you're concerned about delivering emails to your users without slowing down their experience, this is a very easy way to accomplish that.

Chapter 6: How to handle dynamic Subdomains in Symfony

HOW TO HANDLE DYNAMIC SUBDOMAINS IN SYMFONY

From Rafael:

> Hi, Symfony 2.2 has released hostname pattern for urls, I would like to know how can I create a url pattern that match domains loaded from a database? where should I put the code to load the domains and how should I pass this to a routing config file?

And from zaherg:

> How can I handle auto generated subdomains routing with symfony 2?

Answer

Symfony 2.2 comes with hostname handling out of the box, which lets you create two routes that have the same path, but respond to two different sub-domains:

```
homepage:
    path: /
    defaults:
        _controller: QADayBundle:Default:index

homepage_admin:
    path: /
    defaults:
        _controller: QADayBundle:Admin:index
    host: admin.%base_host%
```

The `base_host` comes from a value in `parameters.yml`, which makes this all even more flexible.

But what if you're creating a site that has dynamic sub-domains, where each subdomain is a row in a "site" database table? In this case, the new `host` routing feature won't help us: it's really meant for handling a finite number of concrete subdomains.

So how could this be handled? Let's find out together!

1) The VirtualHost

Before you go anywhere, make sure you have an Apache VirtualHost or Nginx site that sends all the subdomains of your host to your application. Since we're using `lolnimals.l` locally, we'll want `*.lolnimals.l` to be handled by the VHost.

```
<VirtualHost *:80>
  ServerName qaday.l
  ServerAlias *.qaday.l

  DocumentRoot "/Users/leannapelham/Sites/qa/web"
  <Directory "/Users/leannapelham/Sites/qa/web">
    AllowOverride All
    Allow from All
  </Directory>
</VirtualHost>
```

Next, add a few entries to your `/etc/hosts` file for subdomains that we can play with:

```
# /etc/hosts
127.0.0.1       lolnimals.l kittens.lolnimals.l alpacas.lolnimals.l dinos.lolnimals.l
```

Great! Restart or reload your web server and then at least check that you can hit your application from any of these sub-domains. So far our application isn't actually doing any logic with these subdomains, but we'll get there!

2) Create the Site Entity

Next, let's use Doctrine to generate a new `Site` entity, which will store all the information about each individual subdomain:

```
php app/console doctrine:generate:entity
```

Give the entity a name of `QADayBundle:Site`, which uses a `QADayBundle` that I already created. For fields, add one called `subdomain` and two others called `name` and `description`, so we at least have some basic information about this site.

Note

Press tab to take advantage of the command autocompletion. This is the brand new2.2 autocomplete feature in action.

Finish up the wizard then immediately create the database and schema. Be sure to customize your `app/config/parameters.yml` file first:

```
php app/console doctrine:database:create
php app/console doctrine:schema:create
```

Finally, to make things interesting, I'll bring in a little data file that will add two site records into the database:

```
// load_sites.php
require __DIR__.'/vendor/autoload.php';

use Symfony\Component\HttpFoundation\Request;
$loader = require_once __DIR__.'/app/bootstrap.php.cache';
require_once __DIR__.'/app/AppKernel.php';
$kernel = new AppKernel('dev', true);
$request = Request::createFromGlobals();
$kernel->boot();
$container = $kernel->getContainer();
$container->enterScope('request');
$container->set('request', $request);

// start loading things
use KnpU\QADayBundle\Entity\Site;

/** @var $em \Doctrine\ORM\EntityManager */
$em = $container->get('doctrine')->getManager();
$em->createQuery('DELETE FROM QADayBundle:Site')->execute();

$site1 = new Site();
$site1->setSubdomain('kittens');
$site1->setName('Cute Kittens');
$site1->setDescription('I\'m peerrrrfect!');

$site2 = new Site();
$site2->setSubdomain('alpacas');
$site2->setName('Funny Alpacas');
$site2->setDescription('Alpaca my bags!');

$em->persist($site1);
$em->persist($site2);
$em->flush();
```

A better way to do this is with some real fixture files, but this will work for now. This script bootstraps Symfony, but then lets us write custom code beneath it. If you're curious about this script or fixtures, check out our Starting in Symfony2 series where we cover all this goodness and a ton more.

Execute the script from the command line.

```
php load_sites.php
```

I'll use the built-in `doctrine:query:sql` command to double-check that things work.

```
php app/console doctrine:query:sql "SELECT * FROM Site"
```

Great, let's get to the good stuff!

3) Finding the current Site the "Easy" Way

Because of our VirtualHost, our application already responds to every subdomain of `lolnimals.l`. The goal in our code is to be able to determine, based on the host name, which Site record in the database is being used.

First, let's use a homepage route and controller that I've already created. This will seem simple, but for now, let's determine which Site record is being used by querying directly here. I'll add the `$request` as an argument to the method to get the request object, then use `getHost` to grab the host name. Dump the value to see that it's working:

```
// src/KnpU/QADayBundle/Controller/DefaultController.php

use Symfony\Component\HttpFoundation\Request;
// ...

public function indexAction(Request $request)
{
    $currentHost = $request->getHttpHost();
    var_dump($currentHost);die;

    return $this->render('QADayBundle:Default:index.html.twig');
}
```

The value stored in the database is actually *only* the subdomain part, not the whole host name. In other words, we need to transform `alpacas.lolnimals.l` into simply `alpacas` before querying. Fortunately, I've already stored my base host as a parameter in `parameters.yml`:

```
# /app/config/parameters.yml
parameters:
    # ...
    base_host:        qaday.l
```

By grabbing this value out of the container and doing some simple string manipulation, we can get the current subdomain key:

```
// src/KnpU/QADayBundle/Controller/DefaultController.php
// ...

public function indexAction(Request $request)
{
    $currentHost = $request->getHttpHost();
    $baseHost = $this->container->getParameter('base_host');

    $subdomain = str_replace('.'.$baseHost, '', $currentHost);
    var_dump($subdomain);die;

    return $this->render('QADayBundle:Default:index.html.twig');
}
```

Perfect! Now querying for the current Site is pretty easy. We'll also assume that we *need* a valid subdomain - so let's show a 404 page if we can't find the Site:

```
// src/KnpU/QADayBundle/Controller/DefaultController.php
// ...

$site = $this->getDoctrine()
    ->getRepository('QADayBundle:Site')
    ->findOneBy(array('subdomain' => $subdomain))
;
if (!$site) {
    throw $this->createNotFoundException(sprintf(
        'No site for host "%s", subdomain "%s"',
        $baseHost,
        $subdomain
    ));
}
```

Finally, pass the `$site` into the template so we can prove we're matching the right one:

```
// src/KnpU/QADayBundle/Controller/DefaultController.php
// ...

return $this->render('QADayBundle:Default:index.html.twig', array(
    'site' => $site,
));
```

Dump some basic information out in the template to celebrate:

```
{# src/KnpU/QADayBundle/Resources/views/Default/index.html.twig #}
{%  extends '::base.html.twig' %}

{% block body %}
    <h1>Welcome to {{ site.name }}</h1>

    <p>{{ site.description }}</p>
{% endblock %}
```

Ok, try it out! The `alpacas` and `kittens` subdomains work perfectly, and the `dinos` subdomain causes a 404, since there's no entry in the database for it.

This is simple and functional, but let's do better!

4) The Site Manager

We've met our requirements of dynamic sub-domains, but it's not very pretty yet. We'll probably need to know what the current Site is all over the place in our code - in every controller and in other places like services. And we certainly don't want to repeat all of this code, that would be crazy!

Let's fix this, step by step. First, create a new class called `SiteManager`, which will be responsible for always knowing what the current Site is. The class is very simple - just a property with a get/set method:

```
// src/KnpU/QADayBundle/Site/SiteManager.php
namespace KnpU\QADayBundle\Site;

use KnpU\QADayBundle\Entity\Site;

class SiteManager
{
    private $currentSite;

    public function getCurrentSite()
    {
        return $this->currentSite;
    }

    public function setCurrentSite(Site $currentSite)
    {
        $this->currentSite = $currentSite;
    }
}
```

Next, register this as a service. If services are a newer concept for you, we cover them extensively in Episode 3 of our Symfony2 Series. I'll create a new `services.yml` file in my bundle. The actual service configuration

couldn't be simpler:

```
# src/KnpU/QADayBundle/Resources/config/services.yml
services:
    site_manager:
        class: KnpU\QADayBundle\Site\SiteManager
```

This file is new, so make sure it's imported. I'll import it by adding a new `imports` entry to `config.yml`:

```
# app/config/config.yml
imports:
    # ...
    - { resource: "@QADayBundle/Resources/config/services.yml" }
```

Sweet! Run `container:debug` to make sure things are working:

```
php app/console container:debug | grep site
```

```
site_manager   container KnpU\QADayBundle\Site\SiteManager
```

Perfect! So.... how does this help us? First, let's set the current site on the `SiteManager` from within our controller:

```
// src/KnpU/QADayBundle/Controller/DefaultController.php
// ...

/** @var $siteManager \KnpU\QADayBundle\Site\SiteManager */
$siteManager = $this->container->get('site_manager');
$siteManager->setCurrentSite($site);

return $this->render('QADayBundle:Default:index.html.twig', array(
    'site' => $siteManager->getCurrentSite(),
));
```

Don't let this step confuse you, because it's pretty underwhelming. This sets the current site on the `SiteManager`, which we use immediately to pass to the template. If this looks kinda dumb to you, it is! Getting the current site from the `SiteManager` is cool, but the problem is that we still need to set this manually.

In other words, the `SiteManager` is only one piece of the solution. Now, let's add an event listener to fix the rest.

5) Determining the Site automatically with an Event Listener

Somehow, we need to be able to move the logic that determines the current Site out of our controller and to some central location. To do this, we'll leverage an event listener. Again, if this is new to you, we cover it in Episode 3 of our Symfony2 Series.

First, create the listener class, let's call it `CurrentSiteListener` and set it to have the `SiteManager` and Doctrine's `EntityManager` injected as dependencies. Let's also inject the `base_host` parameter, we'll need it here as well:

```php
// src/KnpU/QADayBundle/EventListener/CurrentSiteListener.php
namespace KnpU\QADayBundle\EventListener;

use KnpU\QADayBundle\Site\SiteManager;
use Doctrine\ORM\EntityManager;

class CurrentSiteListener
{
    private $siteManager;

    private $em;

    private $baseHost;

    public function __construct(SiteManager $siteManager, EntityManager $em, $baseHost)
    {
        $this->siteManager = $siteManager;
        $this->em = $em;
        $this->baseHost = $baseHost;
    }
}
```

The goal of this class is to determine and set the current site at the very beginning of every request, before your controller is executed. Create a method called `onKernelRequest` with a single `$event` argument, which is an instance of `GetResponseEvent`:

```php
// src/KnpU/QADayBundle/EventListener/CurrentSiteListener.php

// ...
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class CurrentSiteListener
{
    // ...

    public function onKernelRequest(GetResponseEvent $event)
    {
        die('test!');
    }
}
```

Tip

The Symfony.com documentation has a full list of the events and event objects in theHttpKernel section.

Before we fill in the rest of this method, register the listener as a service and tag it so that it's an event listener on the `kernel.request` event:

```yaml
services:
    # ...

    current_site_listener:
        class: KnpU\QADayBundle\EventListener\CurrentSiteListener
        arguments:
            - "@site_manager"
            - "@doctrine.orm.entity_manager"
            - "%base_host%"
        tags:
            -
                name: kernel.event_listener
                method: onKernelRequest
                event: kernel.request
```

And with that, let's try it! When we refresh the page, we can see the message that proves that our new listener is being called early in Symfony's bootstrap.

With all that behind us, let's fill in the final step! In the `onKernelRequest` method, our goal is to determine and set the current site. Copy the logic out of our controller into this method, then tweak things to hook up:

```
public function onKernelRequest(GetResponseEvent $event)
{
    $request = $event->getRequest();

    $currentHost = $request->getHttpHost();
    $subdomain = str_replace('.'.$this->baseHost, '', $currentHost);

    $site = $this->em
        ->getRepository('QADayBundle:Site')
        ->findOneBy(array('subdomain' => $subdomain))
    ;
    if (!$site) {
        throw new NotFoundHttpException(sprintf(
            'No site for host "%s", subdomain "%s"',
            $this->baseHost,
            $subdomain
        ));
    }

    $this->siteManager->setCurrentSite($site);
}
```

The differences here are a bit subtle. For example, the `baseHost` is now stored in a property and we can get Doctrine's repository through the `$em` property. We've also replaced the `createNotFoundException` call by instantiating a new `NotFoundHttpException` instance. The `createNotFoundException` method lives in Symfony's base controller. We don't have access to it here, but this is actually what it really does behind the scenes.

Since we've registered this as an event listener on the `kernel.request` event, this method will guarantee that the `SiteManager` has a current site before our controller is ever executed. This means we can get rid of almost all of the code in our controller:

```
public function indexAction()
{
    /** @var $siteManager \KnpU\QADayBundle\Site\SiteManager */
    $siteManager = $this->container->get('site_manager');

    return $this->render('QADayBundle:Default:index.html.twig', array(
        'site' => $siteManager->getCurrentSite(),
    ));
}
```

Try it out! Sweet, it still works! We can now use the `SiteManager` from anywhere in our code to get the current Site object. For example, if we needed to load all the blog posts for only this Site, we could grab the current Site then create a query that returns only those items. Basically, from here, you can be dangerous!

Chapter 7: Symfony2: Make my Controllers Services?

SYMFONY2: MAKE MY CONTROLLERS SERVICES?

From Christian:

> Hi,
>
> I'd like to know what you think about the practice of building "controllers as a service" as suggested here:
>
> http://pooteeweet.org/blog/1947
>
> https://github.com/symfony/symfony-docs/issues/457
>
> Thanks! And keep up the great work!

Answer

This is a big religious topic in the Symfony2 community, and if you scan the comments in the links above, you'll see why. In fact, it's not something I usually talk about: it can be a hornet's nest :). So here we go!

In a moment, we're going to walk through an example and compare the approaches. But first, I'll say that I **don't register my controllers as services**, and the reasons behind this are simple:

1) Registering a controller as a service is more work. That's not the worst things ever, but since it takes longer, the rewards need to outweigh this.

2) All of your logic should be pushed out into your service layer anyways. This is the age-old skinny controllers best-practice.

3) And now that your controllers are skinny, there's no need to unit test them. Instead unit test the services being used by your controllers.

4) Services used by your controller are loaded lazily. This is not the case if you've registered your controllers as a service and inject only what you need. But in theory, as long as you keep your controllers *focused*, then what you're injecting will need to be used for any action anyways.

With that viewpoint, the slight increase in setup time probably doesn't make registering your controllers as services worth it. And when we're teaching beginners, it would be yet another concept to need to know early-on.

But as you dive in deeper, the topic gets more complex and the advantages more fascinating, especially for seasoned developers that can register a service very quickly.

A Case for Services

The advantages to registering your controllers as services are more subtle but compelling!

Let's build two controllers so we can compare each in detail.

Injecting the Container - without the Base Controller

The routing for the first looks normal:

```
controller_container:
    path: /controller/container
    defaults:
        _controller: QADayBundle:Container:index
```

Next, let's look at the controller class itself:

```php
// src/KnpU/QADayBundle/Controller/ContainerController.php
namespace KnpU\QADayBundle\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;

class ContainerController implements ContainerAwareInterface
{
    private $container;

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

    public function indexAction(Request $request)
    {
        if ($request->isMethod('POST')) {
            // .. do some things

            $url = $this->container->get('router')->generate('homepage');
            return new RedirectResponse($url);
        }

        return $this->container->get('templating')->renderResponse(
            'QADayBundle::controllerTest.html.twig',
            array('type' => 'Injecting the container!')
        );
    }
}
```

In your Symfony2 projects, you're probably used to inheriting Symfony2's base Controller class. This gives you shortcut methods and makes sure that Symfony's container is set on a `container` property. To see what's really happening, I've chosen *not* to extend this class. Instead, by implementing `ContainerAwareInterface`, we can still make sure that Symfony calls `setContainer` and passes it to us. After that, we grab services directly from the container and use them. This is all exactly what happens behind-the-scenes in your controllers when you extend Symfony's base Controller class.

Creating a Controller as a Service

Next, let's create that same controller, except register it as a service and only inject what we need. First, the routing:

```yaml
controller_service:
    path: /controller/service
    defaults:
        _controller: qa_day.controller.service:indexAction
```

Notice the `_controller` key looks different. We haven't yet, but in a moment we'll create a new service called `qa_day.controller.service`. Notice that we **do** include the `Action` suffix with the method name: when you refer to a controller as a service, none of the normal conventions are assumed (i.e. `index` => `indexAction`).

Next, the actual controller class:

```php
namespace KnpU\QADayBundle\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Bundle\FrameworkBundle\Templating\EngineInterface;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

class ServiceController
{
    private $templating;

    private $router;

    public function __construct(EngineInterface $templating, UrlGeneratorInterface $router)
    {
        $this->templating = $templating;
        $this->router = $router;
    }

    public function indexAction(Request $request)
    {
        if ($request->isMethod('POST')) {
            // .. do some things

            $url = $this->router->generate('homepage');
            return new RedirectResponse($url);
        }

        return $this->templating->renderResponse(
            'QADayBundle::controllerTest.html.twig',
            array('type' => 'Container as a service!')
        );
    }
}
```

The class is perfectly straightforward: we need the `templating` and `router` services, so we inject them. For extra-credit, I've type-hinted the interface for each of these. Now, instead of referencing the `router` through the `container`, we can just reference it directly. You can't see it here, but my IDE is also giving me auto-completion on the `templating` and `router` objects - that's one major advantage.

Tip

Knowing which interface to use for a specific service is not always easy. For example, how did I know to use `EngineInterface` for the `templating` service? If you're not sure what to use, just look for the service in `container:debug` and use the actual class name - not interface - that is used for the service. To see if there's an interface, open that class up and check for it. This isn't a science, but it's a good path to learn more about the interfaces that are actually behind things.

Finally, we have to do the *extra* step: defining the controller as a service:

```yaml
# src/KnpU/QADayBundle/Resources/config/services.yml
services:
    qa_day.controller.service:
        class: KnpU\QADayBundle\Controller\ServiceController
        arguments: ["@templating", "@router"]
```

This is a totally normal and underwhelming service, but it completes the equation. The `qa_day.controller.service:indexAction` value used for the `_controller` key of our route tells Symfony to grab this service and then execute `indexAction`.

Note

Make sure this `services.yml` file is being imported, either by using an`imports` key in `app/config/config.yml` or via a Dependency Injection Extension class (see Episode 3 for more on this).

Comparing the two approaches: A case for Services

Since we've already talked about why you might *not* register a controller as as service, let's explore the advantages of using services. Many of these are summarized from Lukas' blog and comments:

1) Since you're not injecting the whole container, this is an opportunity to **document what your controller does and doesn't do**. When the controller is a service, it's obvious at a glance that it generates URLs and renders templates. We also know that it doesn't talk to the database, send emails, or do anything else.

To make this even cooler, Lukas points out that if you use the JMSDebugginBundle, then you can use the profiler tool to get a clear vision of what parts of your code - including dependencies - make use of a particular service [screenshot]. That's quite powerful.

2) Injecting specific services gives you **auto-completion and clarity on exactly what types of objects you have**. When you reference the services through the container, you don't *really* know what type of object you'll get out. I commonly work around this by creating a private getter function which tells my IDE what to expect:

```
/**
 * @return \Symfony\Component\Routing\Generator\UrlGeneratorInterface
 */
private function getRouter()
{
    return $this->container->get('router');
}
```

Still, if we gain some time by not registering our controller as a service, it's fair to say that we lose some time doing things like this. It's also technically possible that someone in our code changes the `router` to return something that does **not** implement `UrlGeneratorInterface`. In the service controller, PHP would throw a very clear error if this ever happened. In the container controller, the error would be less clear.

3) How much should your controller do? When you inject the entire container, you could potentially have controllers that control many pages that do many different things. As Kris points out, this is much harder if your controller is a service, since eventually you'll be injecting 100 different dependencies. This is a natural way to **make sure controllers stay focused**.

To Service or not Service?

Since not taking a side is lame, I'll pick my winner. But the true answer is that the best approach depends on who you are and your project.

For most people, **don't register your controllers as services**. It's simpler, faster to develop, and avoids non-lazily-loaded service concerns.

So who should register controllers as services? If your team is very comfortable with service-oriented-architecture and your project is quite large, where it's a challenge to keep track of what pieces affect other pieces, then it starts to make more sense. Like with a lot of things in technology, by choosing this path you're asking to handle more complexity but understand that the advantageous for you outweigh that concern.

Phew, ok, have fun!

Chapter 8: How to compile .less styles into .css (on any OS)

HOW TO COMPILE .LESS STYLES INTO .CSS (ON ANY OS)

From dextervip

> Hi, Less language have been growing up a lot but How can I configure assetic manager to compile less css and rewrite it properly in windows environment?

Answer

Note

Special thanks to our very-own Roman on this answer!

We use less in our projects and love it. However, we do have a mixture of operating systems and also had our own issues getting less to compile properly.

Less is typically compiled by `lessc`, which is installed from `npm` (Node Package Manager), which is a part of Node.js. Phew! Now, none of this is necessarily complicated, but if you're not familiar with node and node modules, then it can be a blocker. As the question suggested, this is sometimes even harder on Windows. In fact, Rafael - who asked this question - has his own problems with exactly this.

So what's the solution? Our advice: avoid the problem.

What we mean is to avoid the true less and instead use lessphp - a pure PHP implementation of the less compiler. Normally, I'm a proponent of letting other languages do things they're good at, but if you're having issues with normal less, take advantage of this tool. As an added bonus, `lessphp` has a built-in filter in Assetic, so using it is simple.

Note

While lessphp is very good, nothing is as good as the real thing and it's possible that you'll write vali`less` code that doesn't compile correctly. However, these seem to be edge-cases, so worry about that when it happens.

To install `lessphp`, just add it to your `composer.json` file under the `require` key:

```
"leafo/lessphp": "~0.3"
```

Tip

Curious about the `~0.3` version? It's roughly equivalent to `>=0.3,<1.0` and is awesome. See Package Versions for more details.

Next, configure the `assetic` key on `config.yml` to activate the filter:

```
# app/config/config.yml
# ...

assetic:
    filters:
        lessphp:
            file: %kernel.root_dir%/../vendor/leafo/lessphp/lessc.inc.php
            apply_to: "\.less$"
```

Tip

Unlike most libraries we bring in via Composer, this one does*not* follow the PSR-0 standard, and actually just contains a single (useful) file. The `file` key under assetic filters is built to handle this: the file is required before the filter is used.

Finally, setup the stylesheets in your base layout (or wherever):

```
{# app/Resources/view/base.html.twig #}
{# ... #}

{% stylesheets filter='lessphp' output='css/main.css'
    'bundles/qaday/less/main.less'
%}
    <link href="{{ asset_url }}" type="text/css" rel="stylesheet" media="all" />
{% endstylesheets %}
```

Tip

You only need either the `apply_to` in `config.yml` *or* the `filter='lessphp'` in your template, but not both! With the `apply_to` option, the filter is automatically applied to all `*.less` files.

Woh! That's it! Assuming you have the `use_controller` setting on in `config_dev.yml`, you can just access your page to see it working. In the background, the `main.less` file is being processed and the end-CSS is being returned.

You can also dump your assets and see a shiny-new `main.css` file come out:

```
php app/console assetic:dump --env=prod
```

If you ever have any weird issues - especially when playing with your `assetic` configuration in `config.yml`, try clearing your Symfony *and* browser cache. You don't normally need to do this, but there are some edge cases in this area where you might need to.

Tip

If your CSS files begin to load slowly in the `dev` environment, you may consider turning the `use_controller` setting to `false` and dumping your assets manually with the `--watch` flag. See Starting in Symfony2 Episode 4

Chapter 9: Custom Validation, Callback and Constraints

CUSTOM VALIDATION, CALLBACK AND CONSTRAINTS

From Rafael:

> Hi, I am coding one events calendar, It is adding events however how can I validate if the event I am placing does not conflict time with another one event? I was thinking about entity validation callback but should it be in entity? or repository? I don't want to lose symfony validation that display errors on the forms

Answer

This is a great question because it touches on a few interesting and related concepts: custom validation, assigning errors, and the best practices around all of this.

Let's follow along with your example. Suppose we have an `Event` entity that looks like this (with some extras, like getter and setter methods):

```
// src/KnpU/QADayBundle/Entity/Event.php
namespace KnpU\QADayBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="KnpU\QADayBundle\Entity\EventRepository")
 */
class Event
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /** @ORM\Column(name="name", type="string", length=255) */
    private $name;

    /** @ORM\Column(name="startDate", type="datetime") */
    private $startDate;

    /** @ORM\Column(name="endDate", type="datetime") */
    private $endDate;

    // ...
}
```

I also have a really basic route, controller and form setup which allows the user to create a new Event (check out the code download to see this). Ok, let's get to work!

The Callback Constraint

The goal is to throw a validation error if the event will conflict with the start and end times of some existing event. There are a few ways to add custom validation, including the Callback constraint, which executes an arbitrary method in your model/entity class and lets you apply any custom logic you want:

```php
// src/KnpU/QADayBundle/Entity/Event.php
// ...
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\Validator\ExecutionContextInterface;

/**
 * @Assert|Callback(methods={"checkCustomValidation"})
 */
class Event
{
    // ...

    public function checkCustomValidation(ExecutionContextInterface $context)
    {
        $context->addViolationAt('name', 'Pick a cooler name!');
    }
}
```

This is my favorite way to handle custom validation because it's so easy. The problem is that the method lives in your entity. This means that you don't have access to the entity manager or any other services. In this case, there's no way to query to see if any other event has a conflicting date.

A bit Ugly, but Easy: Callback + constraints

Normally, we add validation constraints to our model class (i.e. `Event`). However, as of Symfony 2.1, additional constraints can be added directly to the form key using a `constraints` option. Like with annotations, you can apply constraints to the whole object, or individual properties.

For simplicity, I've built my form in the controller instead of using a form type class. Let's re-use the `Callback` validator, but now tell it to execute a method on my controller when called:

```php
// src/KnpU/QADayBundle/Controller/EventController.php

use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\Validator\ExecutionContextInterface;
use KnpU\QADayBundle\Entity\Event;
// ...

public function newAction(Request $request)
{
    $form = $this->createFormBuilder(null, array(
        'data_class' => 'KnpU\QADayBundle\Entity\Event',
        'constraints' => array(
            new Assert\Callback(array($this, 'validateEventDates'))
        )
    ))
        ->add('name', 'text')
        ->add('startDate', 'datetime')
        ->add('endDate', 'datetime')
        ->getForm()
    ;

    // ...
}
```

And for now, I've just put some dummy code into the `validateEventDates` function, which lives right inside this same class:

```php
// src/KnpU/QADayBundle/Entity/EventController.php
public function validateEventDates(Event $event, ExecutionContextInterface $context)
{
    $context->addViolationAt('startDate', 'There is already an event during this time!');
}
```

Phew! Let's walk through this step-by-step:

1) We eventually want to validate our object based on multiple pieces of data (the `startDate` and `endDate`). So instead of applying a validator to a single field, we apply it to the whole object. This means that when the `validateEventDates` is called, the whole `Event` object is passed to it.

2) To attach validation constraints directly to the form, we use the `constraints` key and create a new instance of the constraint. Whether you realized it or not, all those `Callback`, `NotBlank`, etc keys that you use every day for

validation are each a real class.

3) When the `Callback` constraint is executed, it detects that we're no longer inside the `Event` class. To help us out, it now passes our method two arguments: the `Event` object and the execution context.

Note

The `Callback` constraint - or any other constraint - can also be applied to just an individual field by adding a third argument to the `add` function, which would be an array with a `constraints` key.

Tip

If your form lives in a form type class, simply add the `constraints` key to the `setDefaulOptions` method.

This solution is a bit ugly because it lives in our Controller, so we can't re-use it or unit test it. We'll improve that in a second, but let's get it working first!

Applying the Validation Logic

Now that the callback method lives in the controller, we can easily access the entity manager (or any other service) and run the queries we need to. And since we are going to be executing some queries, the best place for that logic is in the `EventRepository` class:

```
// src/KnpU/QADayBundle/Entity/EventRepository.php
namespace KnpU\QADayBundle\Entity;

use Doctrine\ORM\EntityRepository;

class EventRepository extends EntityRepository
{
    public function findOverlappingWithRange(\DateTime $startDate, \DateTime $endDate)
    {
        $qb = $this->createQueryBuilder('e');

        return $qb->andWhere('e.startDate < :endDate AND e.endDate > :startDate')
            ->setParameter('startDate', $startDate)
            ->setParameter('endDate', $endDate)
            ->getQuery()
            ->execute()
        ;
    }
}
```

Great! Now use this function in the callback method in the controller:

```
// src/KnpU/QADayBundle/Controller/EventController.php
public function validateEventDates(Event $event, ExecutionContextInterface $context)
{
    $conflicts = $this->getDoctrine()
        ->getRepository('QADayBundle:Event')
        ->findOverlappingWithRange($event->getStartDate(), $event->getEndDate())
    ;

    if (count($conflicts) > 0) {
        $context->addViolationAt(
            'startDate',
            'There is already an event during this time!'
        );
    }
}
```

Tip

If this method lives in your form type class, then you don't have the entity manager! One option is to pass it in as an option when creating your form:

```
$form = $this->createForm(new EventType, null, array(
    'em' => $this->getDoctrine()->getManager()
))
```

The `em` option is then available in the `buildForm` method of the form type class:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $em = $options['em'];
}
```

For this to work, make sure to add `em` to the "defaults" in your form type's `setDefaultOptions` method.

If you try it, it works! It's a bit dirty, but at least our query logic lives in `EventRepository`. If you were also handling "edits", you'd also need to make sure that the result isn't the exact object being saved. But I'll leave that to you!

Creating a Proper Custom Validation Constraint

There's nothing wrong with what we have so far, but for the sake of reusability, clean code and unit testing, it can be much better.

The ultimate solution to custom validation is to create your own constraint. Fortunately, we've already done most of the work. Start by creating a new `UniqueEventDate` class:

```
// src/KnpU/QADayBundle/Validator/UniqueEventDate.php
namespace KnpU\QADayBundle\Validator;

use Symfony\Component\Validator\Constraint;

/** @Annotation */
class UniqueEventDate extends Constraint
{
    public function validatedBy()
    {
        return 'unique_event_date';
    }

    public function getTargets()
    {
        return self::CLASS_CONSTRAINT;
    }
}
```

Yep, this class is so simple it's silly. Each custom validation constraint is actually two classes: one "Constraint" (seen here) that holds some options and another "Constraint Validator" (shown next) which does all the work. In fact, you can find these for the built-in constraints, for example `NotBlank` and `NotBlankValidator`.

There are 3 interesting parts to this class:

1) The `@Annotation` will eventually allow us to reference this constraints in the Event class via, well, annotations.

2) The `validatedBy` tells Symfony about the "Constraint Validator" that will actually do the heavy lifting. The `unique_event_date` string shouldn't make sense yet - but it'll be more obvious in a minute.

3) The `getTargets` method defines whether this constraint can be applied to an entire class, a property, or both. Again, since we need multiple values on `Event` in order to make our validation decision, we will apply the constraint to the entire class.

Tip

This example doesn't use any constraint options. If you do want to see what it looks like to have a constraint that has configurable options, see the core Email and EmailValidator classes.

Next, create the "Constraint Validator" class:

```php
// src/KnpU/QADayBundle/Validator/UniqueEventDateValidator.php
namespace KnpU\QADayBundle\Validator;

use Symfony\Component\Validator\ConstraintValidator;
use Doctrine\ORM\EntityManager;
use Symfony\Component\Validator\Constraint;

class UniqueEventDateValidator extends ConstraintValidator
{
    private $em;

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

    public function validate($object, Constraint $constraint)
    {
        die('hold on, we\'ll fill finish this in a second...');
    }
}
```

In a second, we'll fill this class in and have it do all the validation work. But first, register it as a service and tag it with a special validator.constraint_validator tag:

```yaml
# src/KnpU/QADayBundle/Resources/config/services.yml
services:
    unique_event_date_validator:
        class: KnpU\QADayBundle\Validator\UniqueEventDateValidator
        arguments:
            - "@doctrine.orm.entity_manager"
        tags:
            -
                name: validator.constraint_validator
                alias: unique_event_date
```

Note

Make sure this `services.yml` file is being imported, either by using an`imports` key in `app/config/config.yml` or via a Dependency Injection Extension class (see Episode 3 for more on this).

Notice that the `alias` we use with the tag corresponds with the value that the Constraint class returns in `validateBy`. This is how Symfony knows that the `UniqueEventDateValidator` is the real muscle behind the `UniqueEventDate` constraint.

Ok! Before we fill in the logic in the `validate` method, let's try this out! The new constraint isn't magically activated - we activate it like any other constraint, with annotations (or YAML, if you prefer):

```php
// src/KnpU/QADayBundle/Entity/Event.php
// ...

use KnpU\QADayBundle\Validator\UniqueEventDate;

/**
 * @ORM\Entity(repositoryClass="KnpU\QADayBundle\Entity\EventRepository")
 * @UniqueEventDate()
 */
class Event
{
    // ...
}
```

When you submit the form, the `UniqueEventDate` constraint is triggered, and ultimately the `UniqueEventDateValidator::validate` method is called. In other words, you'll see our `die` statement print.

Ok, let's finish this! Copy the logic from the controller `validateEventDates` method and remove it and the `constraints` option while you're there. Paste it into `UniqueEventDateValidator::validate` and adjust it accordingly:

```
// src/KnpU/QADayBundle/Validator/UniqueEventDateValidator.php
public function validate($object, Constraint $constraint)
{
    $conflicts = $this->em
        ->getRepository('QADayBundle:Event')
        ->findOverlappingWithRange($object->getStartDate(), $object->getEndDate())
    ;

    if (count($conflicts) > 0) {
        $this->context->addViolationAt('startDate', 'There is already an event during this time!');
    }
}
```

Let's walk through the differences:

1) Since we've injected Doctrine's Entity Manager, we can access it and get the `EventRepository` through `$this->em`.

2) Since we applied the `UniqueEventDate` constraint to the `Event` class, the entire `Event` object is passed as the first argument to this method (i.e. `$object`).

3) The `ExecutionContext` is stored automatically on the `$this->context` property.

That's it! When you re-submit the form, the `UniqueEventDate` constraint on `Event` activates this method, which does all the work.

Through all of this, one nice thing is that we were always in complete control of which field our error was attached to. I chose to attach the error to the `startDate` field, but you can use whatever makes sense to you. If you use the `addViolation` method instead, the error will be attached to the whole form and displayed at the top:

```
$this->context->addViolation('There is already an event during this time!');
```

Ok, start validating!

Chapter 10: How to (dynamically) remove a Form Field

HOW TO (DYNAMICALLY) REMOVE A FORM FIELD

From ThiagoKrug:

> Hi, I am reusing one form but I have one specific controller action that I need remove one field form, How can I do it without creating new form? Thanks!

Answer

Note

Special thanks to our very-own Roman on this answer!

Cool question! There are actually multiple ways to achieve this task, ranging from a very simple `->remove()` method call to setting up a form event listener.

Let's review the two easiest and most common ways.

First, initialize a simple form type with two fields:

```
// src/KnpU/QADayBundle/Form/Type/RemoveFormFieldType.php
namespace KnpU\QADayBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class RemoveFormFieldType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('first', 'text')
            ->add('second', 'text')
        ;
    }

    public function getName()
    {
        return 'remove_form_field';
    }
}
```

Next, let's build the two different controllers that will render this form to show off the two solutions:

```php
// src/KnpU/QADayBundle/Controller/RemoveFormFieldController.php
namespace KnpU\QADayBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use KnpU\QADayBundle\Form\Type\RemoveFormFieldType;

class RemoveFormFieldController extends Controller
{
    public function firstWayAction()
    {
        $form = $this->createForm(new RemoveFormFieldType());

        // form processing...

        return $this->render('QADayBundle:RemoveFormField:form.html.twig', array(
            'form' => $form->createView()
        ));
    }

    public function secondWayAction()
    {
        // same as firstWayAction() for now
    }
}
```

Awesome! Now, let's solve this in 2 different ways.

Option 1: Using remove

The most straightforward way to achieve this is by removing the form field you want with the `remove` function:

```php
public function firstWayAction()
{
    $form = $this->createForm(new RemoveFormFieldType());

    $form->remove('second');

    // form processing...

    return $this->render('QADayBundle:RemoveFormField:form.html.twig', array(
        'form' => $form->createView()
    ));
}
```

And that's it! No changes to `RemoveFormFieldType` are required at all! And the best part is that this is a perfectly valid solution, no need to over-think it :).

Option 2: Using Form Options

But if you want a more advanced solution with a bit more flexibility, there's another way!

First, let's tweak the form type a bit to rely on a custom option passed on initialization:

```php
// src/KnpU/QADayBundle/Controller/RemoveFormFieldController.php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
// ...

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('first', 'text');

    if ($options['use_second']) {
        $builder->add('second', 'text');
    }
}

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'use_second' => true
    ));
}
```

Now, when you create the form, pass the option and you're ready to go!

```php
public function secondWayAction()
{
    // the "null" option is the form data - you might pass something here
    $form = $this->createForm(new RemoveFormFieldType(), null, array(
        'use_second' => false
    ));

    return $this->render('QADayBundle:RemoveFormField:form.html.twig', array(
        'form' => $form->createView()
    ));
}
```

That’s it! There is also an event dispatching/listening system in the Form component, which allows you to dynamically add/remove/modify fields based on anything (e.g. user-submitted data). For more information, see How to Dynamically Modify Forms Using Form Events.

Have fun!

Chapter 11: Symfony2: Setup, Configuration, Rad?

SYMFONY2: SETUP, CONFIGURATION, RAD?

From Andy:

> As Symfony becomes more powerful and scalable, is that at the expense of being less useful for small rapid developments - given the large number of dependencies and configuration required on each project? (Setting up user login, media management, content repository, Gaufrette, admin generator, etc.).

Answer

Woh! This is a tough question, but one that certainly affects us every day.

There are probably a lot of ways to tell the story of RAD versus "enterprise", but let's look at Symfony. Symfony1 was basically a port of early Ruby on Rails: convention over configuration, RAD, etc. The goal was to develop things faster, with less repetition, and the results were revolutionary.

Fast forward 5 years to Symfony2. Short-cuts have been replaced with best-practices and convention replaced with predictability and explicitness.

The end result is that developing a feature may be faster in symfony1 or something like it. Conversely, that feature is probably cleaner and more maintainable inside Symfony2 due to better practices and less shortcuts (leading to less wtf moments).

This isn't the end of the story (keep reading), but it does mean that you need to choose the right tool for the job. If you're the only developer on a small project, it might be better to choose Silex or something smaller. If you're building something more complex, Symfony2 becomes a more clear winner.

RAD Versus Quality? Both?

Symfony1 and other "RAD" PHP frameworks use a lot of bad practices and magic whereas Symfony2 fixes that but is less RAD out-of-the-box.

So, can we have RAD *and* high-quality tools.

The answer is a resounding YES, though we're not totally there yet.

The topic actually came up very recently in a blog post by Lukas Smith called Good design is no excuse for wasting time. Going back to our history lesson, symfony1 may have been RAD, but its architecture was fundamentally flawed and coupled. Fixing it meant re-building correctly form the ground-up. This doesn't mean that we *can't* also be RAD, it just means that RAD tools need to be built on top of Symfony2.

And while it's true that there's a lot of integration still to worry about between user management, asset management, Gaufrette, admin areas, etc, you *do* have some options, which Lukas points out:

1. KnpRadBundle

A *lot* of love at Knp has been put into this little project, which takes the Symfony2 framework experience and makes it opinionated. Things are integrated more naturally and there are plenty of shortcuts. But, it's still the Symfony2 framework, so you're not learning something new, just opting into RAD features.

2. Laravel

Laravel4 is being built on top of Symfony2, and while I haven't tried it out yet, my impression is that it lowers the Symfony2 learning curve (and I'm assuming also adds some RAD). This is another great example of taking our new solid core and making it quicker to develop things.

3. Silex

Silex is the micro-framework built on top of Symfony2, which lets you get an application going instantly. It's not suitable for everything, and eventually you'll wish you had more tools, but you'll get started *fast*.

There are several other things, which provide pieces to complete the puzzle (e.g. SonataAdminBundle, FOSRestBundle), but more work certainly needs to be done to bring all of these great pieces together into one, harmonic - and more opinionated - piece. It’s a work-in-progress, but it’s not a reason to *not* choose Symfony2. In my projects, I choose either the Symfony2 framework (usually with KnpRadBundle) or Silex, depending no the complexity of the app. Because they’re both built in the same solid core, the learning curve between the two is basically non-existent.

Happy RAD’ing!

Chapter 12: Complex Symfony2 Examples: Users, Menus, CMS Features

COMPLEX SYMFONY2 EXAMPLES: USERS, MENUS, CMS FEATURES

From pieter lelaona:

> It's been very difficult to find examples of applications and explanations actually implement symfony2 framework. e.g.
>
> 1. how to create a complete multi-user management with an ACL that retrieves data from a database and integrated with the menus link, filter data, and dynamic multi-role and permission
> 2. best project skeleton in symfony2 framework.
> 3. create a dynamic system such as the theme cms wordpress, drupal, joomla, etc
>
> This is a small part of what many people, especially in my country (Indonesia) want to learn more about the Symfony2 framework. what do you think??

Answer

Hi Pieter! As you know, Symfony2 isn't a CMS but contains all the tools needed to create any system of any complexity that you want. However, for complex systems like you're describing, there are ultimately many pieces that need to be integrated to get this all working.

This is a huge topic, but let's go through your questions and clarify the best way to approach each.

1) Multi-user system with ACLs, Menus and Filtering

This is *still* a huge topic, so let's break it down even further:

a. Multi-User systems
b. ACL's
c. Menus
d. Filtering

Multi-User Systems

Creating multi-user systems that load user and permission information from the database is easy in Symfony2. Depending on your preference, you will probably either use the popular FOSUserBundle or implement this yourself by following our How to load Security Users from the Database cookbook entry.

In either case, creating a system with "groups" and "permissions" is very possible, where a user belongs to many groups and each group has a sub-set of permissions. In Symfony's point-of-view, each user ultimately has an array of "roles", which are returned by your User object's `getRoles` function. You can use whatever logic you want to return these, including referencing "groups" and "permissions" database relationships.

In fact, Groups Functionality is available in FOSUserBundle out-of-the-box. This works simply because their base `User` object calculates its roles by aggregating all of the roles (or permissions) across all of the groups:

```php
public function getRoles()
{
    $roles = $this->roles;

    foreach ($this->getGroups() as $group) {
        $roles = array_merge($roles, $group->getRoles());
    }

    // we need to make sure to have at least one role
    $roles[] = static::ROLE_DEFAULT;

    return array_unique($roles);
}
```

You can do the same thing - or whatever complex logic you want - to determine the roles that a user should have.

ACL's

This is a very common question, and my answer might be surprising.

Symfony2 has built-in ACL functionality, which I *never* use. I'm sure it has its use-cases, but each time that I talk to someone that wants to use Symfony's ACL's, what they really need is a *voter*.

What's a voter? I'm glad you asked! First, let's look at one way to enforce security from within a controller:

```php
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function indexAction()
{
    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted('ROLE_USER')) {
        throw new AccessDeniedException('Get outta here!');
    }
}
```

On the surface, `isGranted` simply checks to see if the current user has this role and returns `true` or `false`. But behind the scenes, Symfony passes `ROLE_USER` (called an "attribute") to a number of "voters" and asks each to "vote" on whether or not the current user should be "granted" `ROLE_USER`.

And while it's technically possible for two voters to vote on a single attribute and disagree with each other, life is much simpler in reality. Symfony2 comes with 3 voters by default:

1) RoleVoter Votes only if the attribute starts with `ROLE_` and checks to see if the current user has this exact attribute as a role.

2) RoleHierarchyVoter Votes only if the attribute starts with `ROLE_` and checks to see if the user has this role by using the role hierarchy.

3) AuthenticatedVoter Votes only if the attribute is `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED` or `IS_AUTHENTICATED_ANONYMOUSLY`.

So what happens if we invent a new type of attribute that none of these voters "votes" on?

```php
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function indexAction()
{
    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted('CONTENT_EDIT')) {
        throw new AccessDeniedException('Get outta here!');
    }
}
```

In this case, none of the existing voters will vote on `CONTENT_EDIT`. You won't get an error: `isGranted` will silently return `false` (by default). This is significant - as we'll see in a moment - because we can create our own voters that respond on these new attributes.

One other commonly-unknown property of `isGranted` is that there's a second argument, which is any type of "object":

```php
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function showAction($slug)
{
    $post = // query for a Post object using the $slug

    $securityContext = $this->container->get('security.context');
    if (!$securityContext->isGranted('CONTENT_EDIT', $post)) {
        throw new AccessDeniedException('Get outta here!');
    }
}
```

When you do this, each “voter” is passed the object. This is very important because it means that your custom voter can make its access decision based off of a specific piece of data. This is typically what you think of when you talk about ACL: the ability to say that “this user” has access to “edit” some “object”. In Symfony2, you can leverage a custom voter to use whatever complex business logic you have to determine this.

This is a somewhat shortened version of this topic, but there is a cookbook article on creating voters. However, you’ll do several things differently in your implementation:

- Invent your own attributes - like `CONTENT_EDIT` and `CONTENT_DELETE` and make your voter only respond to those.
- Use the `$object` argument passed to your `vote` function. You may then need to determine what type of object it is (e.g. is this a blog post? A user object?) and use any business rules you have (querying some database relationships) to determine if access should be granted.
- You will not need to change the “Access Decision Strategy”.

I hope this at least gives you some direction on using ACL’s without ACL’s in Symfony2! The big disadvantage to this method is performance. But since the solution is so much more natural than ACL’s, you should worry about this later when it’s an issue. You can always cache the decisions you’re making, which is very similar to what true ACL’s do in the database.

Menus

If you’re building complex menus in Symfony2, then you should be using KnpMenuBundle.

This bundle allows you to build your menus inside a PHP class. This is really important because it means that you can do whatever you want when determining which menu items to show or not show for a user.

Let’s start with example that’s directly from the KnpMenuBundle Documentation:

```php
// src/Acme/DemoBundle/Menu/Builder.php
namespace Acme\DemoBundle\Menu;

use Knp\Menu\FactoryInterface;
use Symfony\Component\DependencyInjection\ContainerAware;

class Builder extends ContainerAware
{
    public function mainMenu(FactoryInterface $factory, array $options)
    {
        $menu = $factory->createItem('root');

        $menu->addChild('Home', array('route' => 'homepage'));
        $menu->addChild('About Me', array(
            'route' => 'page_show',
            'routeParameters' => array('id' => 42)
        ));
        // ... add more children

        return $menu;
    }
}
```

To conditionally show the `About Me` link, we can wrap it in a call to the `isGranted` function:

```
$securityContext = $this->container->get('security.context');
if (!$securityContext->isGranted('ROLE_ADMIN')) {
    $menu->addChild('About Me', array(
        'route' => 'page_show',
        'routeParameters' => array('id' => 42)
    ));
}
```

Remember also that you can use your own custom attributes here that hook up to your own custom voters. There are certainly more complex things beyond this, but it will always mean using your voters to determine which entries should be shown.

Filtering

The last piece of all of this is how we filter data based on the user's permissions. Unfortunately, this works much differently than voters where you start with an object and then determine if the user has some sort of permissions to operate on that object.

One way or another, the solution is one that comes down to writing good repository methods that filter your data properly. For example, suppose that you have a `Post` entity with a ManyToMany relationship to `User` that stores all of the users that have access to edit this blog post:

```
// src/KnpU/QADayBundle/Entity/Post.php
// ...

/**
 * @ORM\ManyToMany(targetEntity="User")
 */
protected $admins;
```

In this case, a custom repository method should be added to `PostRepository` to fetch all of the blog posts that this user can edit:

```
// src/KnpU/QADayBundle/Entity/PostRepository.php
// ...

public function findAllEditableByUser(User $user)
{
    // query for all Post objects that have a Post.admins join to this User
}
```

This can be used from within your controller and a related (more efficient) version could also be used inside your custom voter to determine if a user has access to edit one specific blog post. These two repository functions can share most of their logic to avoid any duplication.

In other words, there's no magic to do all of this, but the solution is quite straightforward. By leveraging well-built repository methods, we can re-use that logic in both our custom voters (when determining if a user has access to do something with an object) and in a controller (to get a list of all the items a user has access to).

2) Best Project Skeleton for Symfony2

Symfony2 uses "distributions", which are like pre-started projects using the Symfony2 framework. In theory, there could be a lot of these, though in practice, there aren't very many that I'm aware of. Your best option is to start with the Symfony Standard Edition, which can be downloaded at Symfony.com.

If you've started a few projects with Symfony, and they always look the same, then you might even create your own distribution. A distribution is nothing more than a Symfony2 "project" at some state. In other words, if you start with the Symfony2 Standard Distribution, delete the AcmeDemoBundle, then install and configure a few bundles that you like, then you've just created your very own project skeleton. This is a great option for people that start a lot of Symfony2 projects.

3) Dynamic systems and themes like a CMS

This is also a huge topic, but we can at least link to various resources related to this.

On the "CMS" side of things (particularly content storage), take a look at the Symfony CMF project. This is not meant to be a CMS - if you need something like a CMS, I recommend using an actual CMS, like Drupal. Instead,

it’s all about standardizing how content is stored.

If you’re looking for “theming” functionality, that’s also very possible in Symfony2 due to its flexibility. One great bundle for this - which may work for you or at least serve as an example - is LiipThemeBundle.

That’s a rushed explanation of a *huge* question, but hopefully it gives you some things to look into!

Cheers!

Chapter 13: Symfony2: Organizing your Business Logic into Models

SYMFONY2: ORGANIZING YOUR BUSINESS LOGIC INTO MODELS

From Audrius

> As Symfony is Request/Response rather than MVC framework what is best (business/developer ratio) structure to implement model layer into Symfony applications.
>
> Lets say you have a lot of business logic inside your application, or porting normal MVC application to Symfony, what is best way (in your opinion) to organize structure for applications? All business logic goes into services? Fat controllers? Any other solutions?

Answer

As Audrius correctly points out, Symfony2 isn't actually an MVC framework, nor does it want to be. Symfony2 is all about converting a "request" into a "response". Behind the scenes it uses a simple routing -> controller setup. Using templates, or creating a rich service-oriented-architecture is totally optional and up to you. You can even create your own classic view layer if you want to.

This means that you have a lot of flexibility on how to organize things. But in my opinion, the answer is simple: **create a service-oriented architecture where all your business logic lives in services**. This means having "skinny" controllers and a "fat" model. There will of course be edge-cases, but this is almost always the best way to organize things.

Tip

If any of this "skinny controllers" and "fat" models is new to you, check out our free Dependency Injection screencast.

But this isn't a hard rule. Having a perfectly-organized service layer is something to strive towards, but not something that's always easy - or even good - in the real world. If you're trying to quickly prototype something, for example, then creating services is probably not as good as putting the logic directly in your controller. In fact, you might even argue that logic should live in the controller unless you're going to unit test it or until you need to re-use it.

In other words, the goal is to put your logic in services. Balance that with the real-world requirements of getting things done quickly to compromise between developing quickly and having clean maintainable code. Adding a lot of logic to your controller is a perfect example of Technical Debt, which is a natural part of the development process.

Cheers!

Chapter 14: Conditionally Requiring a Form Field in Symfony2

CONDITIONALLY REQUIRING A FORM FIELD IN SYMFONY2

From David

> Is there a sane way with the form layer and a custom form type to determine if a field is required based on the actual content that is bound to it? I hacked up this gist which i hope shows the idea:
>
> https://gist.github.com/dbu/5142035

Answer

If you don't know David, he's a fantastic developer who works at Liip and spends a lot of time working with the Symfony CMF project. So, when I saw a question from him, I knew it would be tough! Depending on exactly what you're trying to do, this may or may not have a great solution, but we'll learn a lot about building form fields, events and form configuration along the way.

This question is all about being able to dynamically modify a form field after its already been built. Typically, this is done by using a form event and looks something like this:

```
class AddressType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('country', 'count', array(...))
        ;

        $builder->addEventListener(
            FormEvents::PRE_BIND,
            function(FormEvent $event) use($factory){
                $data = $event->getData();
                $form = $event->getForm();

                $country = $data['country'];
                $form->add('state', 'choice', array(
                    'choices' => array() // build state choices from country
                ))
            }
        );
    }
}
```

Note

This example is a little incomplete. See How to Dynamically Modify Forms Using Form Events.

In this case, the `state` field isn't built initially: it waits until the form data is set and then is built based off of the value of the `country` field.

David's example is a little bit more difficult. In the above example, your "form" is modifying a child field. However in David's example, a field is modifying itself.

To see the problem - and talk about possible and impossible solutions - let's start with a custom form type that extends the built-in `file` type:

```php
// src/KnpU/QADayBundle/Form/Type/ImageType.php
namespace KnpU\QADayBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ImageType extends AbstractType
{
    public function getName()
    {
        return 'my_image';
    }

    public function getParent()
    {
        return 'file';
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'required' => true,
        ));
    }
}
```

This isn't very interesting yet, and defines a new field type that looks and acts just like the normal `file` type. We've made it always default to `required`, which is actually the default behavior.

Making a field conditionally-required

Right now, the field is *always* required. Our goal is to make it only required if the base object that it's attached to is "unsaved". If you imagine we're creating an `Event` that has an image, then the user should be required to upload the image when creating the event, but then not required when editing later.

But first, what exactly does the `required` option do? In fact, it has nothing at all to do with server-side validation, which is handled by an entirely different mechanism (and that would also need to be adjusted to meet our end-goal). The `required` option is used in exactly two places by default:

1) It controls the `required` form view variable, which determines whether or not the HTML5 `required` attribute should be used on the field.

2) It's used in the default implementation of the empty_data option. When a form or field has no data, this option is used to give it data. Typically the empty data is either an empty string or an empty `array()`. But if your field or form has a data_class option, then something different happens. If `required` is true, the "empty data" is a new instance of the object specified in `data_class`. If it's `false`, then your empty data is simply null.

In this example, we don't really care about the second usage (though it's really interesting!): we simply want to prevent the `required` attribute from printing.

The easiest way to do this is by overriding the `buildView` method in your custom field:

```php
// src/KnpU/QADayBundle/Form/Type/ImageType.php
use Symfony\Component\Form\FormView;
use Symfony\Component\Form\FormInterface;
// ...

public function buildView(FormView $view, FormInterface $form, array $options)
{
    if ($form->getParent()->getData()->getId()) {
        // this is not new, so make it not required
        $view->vars['required'] = false;
    }
}
```

But before you run and put this in your project, let's talk about several big assumptions that this makes:

1) This assumes that your field has been added to a form with a `data_class` option. The `$form->getParent()->getData()` would then return that object.

2) This assumes that this parent object has a `getId` function, and that calling it is the correct way of checking whether or not the field should be required.

These may vary in your project, and you might even choose to make them configurable in some way.

A solution that doesn't work: Event Listeners

Let's also talk about one solution that does *not* work in this case: form event listeners. Typically, an event listener is used when you want to modify a form field based on some data - often the underlying data in the form itself. This actually sounds like *exactly* what we want, so let's try a simple example (which is basically taken from the gist mentioned in David's question):

```
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\FormEvent;
// ...

class ImageType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->addEventListener(
            FormEvents::PRE_SET_DATA,
            array($this, 'determineRequired')
        );
    }

    public function determineRequired(FormEvent $event)
    {
        $imageForm = $event->getForm();

        if (!$imageForm->getParent()->getData()->getId()) {
            /** @var $formConfig FormBuilderInterface */
            $formConfig = $imageForm->getConfig();

            $formConfig->setRequired(true);
        }
    }
}
```

Sadly, this does *not* actually work. As soon as you call `setRequired`, you'll see the following error:

> FormConfigBuilder methods cannot be accessed anymore once the builder is turned into a FormConfigInterface instance.

That's a bit technical, and relates to how we configure "form builders" and eventually those are used to create the true "Form" object. In this case, it's just too late to do this. The key difference between this and a normal "form events" example is that this field is trying to modify *itself*, whereas usually an entire form will use an event to modify a child field. It turns out that in practice, this seems to make a huge difference.

But this does at least show a few interesting things about the low-level life of a form. First, many of the options that you pass when building a form field are ultimately available on the final Form object. Often, these are actually stored on a FormConfigInterface object, accessible via `$form->getConfig()`:

```
$config = $form->getConfig();
```

But since this solution doesn't actually work, your best method - unless there's a solution hiding somewhere - is to find out what behavior the `required` option causes, and change that behavior directly. Earlier, we did exactly that by modifying the `required` form view variable which controls the HTML5 `required` attribute.

Happy forming!

Chapter 15: Symfony2 Security, Firewalls and Dinosaurs

SYMFONY2 SECURITY, FIREWALLS AND DINOSAURS

From Gerard Araujo:

> What is a typical/ideal bundle and firewall structure for symfony 2 for a project with the following basic requirements:
>
> - frontend [public]
> - frontend [for logged in]
> - backend [for admin]
>
> ... and a few entities that are owned by users like books, media, category...
>
> 1. assuming i use fosuserbundle, should i have at least 2 bundles (1 for fos extension) ? more than 2? or only 1? what advantage/disadvantage to i get with each option?
>
> 2. is one firewall sufficient? what if i need different login routes?
>
> 3. how many dinosaurs does it take to replace a lightbulb?

Answer

Hi Gerard! Uh oh, a security question!

> ... ryan runs away...

Actually, this should be pretty painless. The security component in Symfony2 sometimes suffers from being so flexible that it's not clear how to configure it. Let's try to clarify a bit.

1) FOSUserBundle and Number of Bundles in my Project

This has nothing to do with security, but is a common question: how many bundles should I have and how do I know when I need to create a new bundle? In this case, Gerard is using FOSUserBundle and is wondering how to organize the bundles in his project. In your project, you *will* need a bundle for "User" functionality like your User entity, templates that override FOSUserBundle templates, etc etc.

As Gerard is eluding to, when you want to override pieces of a vendor bundle, there are typically two strategies

1) Placing files in the `app` directory in a specific organization to override some files from a vendor bundle (http://symfony.com/doc/current/book/templating.html#overriding-bundle-templates)

2. Using bundle inheritance.

In the second strategy, you would create a `UserBundle` (or `AcmeUserBundle` depending on your "vendor" namespace), set its parent to `FOSUserBundle`, then begin overriding things.

But let's step back for a second. On a philosophical level, how many bundles should our project have? 1? 5? 50? The answer - like with anything - is up to you. However, don't fool yourself by thinking that you can separate your features into totally standalone, decoupled bundles. In reality, your bundles will be totally coupled to each other and often times it won't be clear exactly which bundle some piece of functionality should live in. And that's ok! We're building one application with one codebase: not an open-source library.

The point is this: don't create new bundles each time you have a new idea. Try to keep your total number of bundles low, and create a new bundle only wen you feel that things are getting crowded.

In our example, I *would* create a `UserBundle` in my project, because I personally really like the "bundle inheritance" strategy for overriding parts of a vendor bundle. And because I did this, I would put *all* my user stuff in here (I wouldn't create yet another bundle for user stuff that doesn't relate to FOSUserBundle).

Beyond that, it's up to you. You might choose to create only one other bundle and put everything into it or create several other bundles. Just don't go overboard.... trust me!

2) Number of Firewalls

One firewall is enough.

I can say this almost regardless of what your project looks like. We talk a lot about firewalls and organization in Starting in Symfony2 Episode 2 and while there are good use-cases for multiple firewalls, they're not very common. Legitimate reasons include:

1) You only use security for one part of your site, that part of your site lives under a specific URL pattern (e.g. `/admin`), and you're very very worried about the small performance hit that loading the security system will cause on every page outside of this section.

2) You have an API that authenticates in a completely different way than your frontend, user data is loaded from a different source, and the API is also only accessible under a very specific URL pattern (e.g. `/api`).

Having multiple firewalls can cause a lot of extra work and confusion. If you have a "frontend" and an "admin" section, my advice is to have only one firewall, load users all from the same source (e.g. from the same database table), then control access to different users and areas of your sites via roles and access controls. This will make you much happier :).

3) How many dinosaurs does it take to replace a lightbulb?

I watched Jurassic Park last night to research this question, but no light bulbs! But I can say that it only takes one dinosaur to open a door... and then bad things happen.

Cheers!

Chapter 16: Training: The Hardest Part

TRAINING: THE HARDEST PART

From our friend John Kary:

> When conducting trainings, what is the biggest challenges new Symfony2 developers faced, and how have you helped them overcome them?

Answer

One of the most interesting and rewarding parts of my job is traveling around and training developers in Symfony2 and Behat. I've worked with developers from all sort of background - including people new to PHP and people that have used symfony1 for years.

Usually a training lasts for 2-3 days where we build a real project in Symfony2. I walk around, ask leading - or misleading :) - questions, then let the trainees use their own smartness to code, research, and make mistakes.

It's always an awesome experience for everyone, except, for the first half of the first day. The biggest challenge that new developers face is in the first 4 hours of being introduced to Symfony2. It's also - paradoxically - the part where we do the easiest things.

The reason is the sheer number of small things that you learn in those first 4 hours. None of them are hard, but it can be overwhelming:

- Namespaces?
- Composer?
- What is the standard distribution?
- Bundles?
- Remove Acme what? in AppKernel what?
- Why am I editing these random config files? routing_dev.yml? routing.yml?
- What's a route? Why does it live here?
- Why am I creating a `DefaultController` class?
- The app directory? src directory? Bundle directory? Lots of directories!?
- What is this `::base.html.twig` file?
- Wait, so `MyBundle:Default:index` is different than `MyBundle:Default:index.html.twig`?

And for the first 4 hours, you learn these all at once. It's also the time where you see the most "Symfony'isms": things that are perfectly specific to the Symfony framework. For example, while "routing" is a generic concept common to all frameworks, the `MyBundle:Default:index` controller sytnax is totally form Symfony.

The good news is that by the end of the first day, this is all ancient history. By diving in and getting hands-on with the code, we spend the rest of the training peeling the layers off of Symfony, discovering what's really going on, how you can take complete control, and more advanced features.

During the first 4 hours, you might be thinking: "I don't know what's going on, I'm just blindly following these directions". And while I wish this could be easier - learning something new isn't always simple. But with patience and perseverance, we always get through it and come running out the other side.

By the end of a few days, you're bored with Symfony, because you've peeled back all its layers.

And that's really exciting.