What's new in Symfony 2.2 + ESI Fragments Bonus

With <3 from SymfonyCasts

Chapter 1: Woh! 2.2 is here!

UPGRADE!¶

So, we heard you like Symfony, so we're showing you how to put more Symfony in your Symfony! Now that 2.2 has been released, we wanted to take some time to show you how to upgrade and highlight a few of our favorite features. We'll be upgrading our 2.1 events project from part 1 of our Getting Starting in Symfony2 series and experimenting with a few new features, like the new fragments framework, caching, routing changes, console command goodies and more. We'll be using this test page which has an inner box that uses Symfony's `render` and gets its content by executing another controller.

The "Big Picture" of Symfony 2.2¶

Symfony 2.2 is special for a few reasons. First, it breaks very minimal backwards compatibility. This means that you should be able to upgrade your project without throwing your laptop out the window. Second, it was the first release as part of Symfony's new process: 1 release every six months, with time for features to mature, documentation to be updated, and bugs to be fixed. If you love when upgrades cause your application to blow up on production, then this will be a boring release for you. If you want some great new features, then stay tuned.

Chapter 2: Upgrading to 2.2

UPGRADING TO 2.2¶

To find details about this release, the actual blog post about it is a great spot. Upgrading a Symfony2 project actually means upgrading the libraries in your *vendor/* directory. The code in your project is, well, *your* code. Symfony2 and all its friends just sit there in vendor/ and wait for you to use them.

Updating composer.json¶

The blog post contains a diff of how your composer.json needs to change. An easier way to see this for any release is to go to the Symfony Standard Edition Repository and find the exact tag you want. Right now, 2.2.0 is the latest, so I'll select it by clicking on the hash. From here, we can browse how the code looked at the moment of this release. And so when we open *composer.json*, we're seeing exactly how it should look for this version:

```
{
    [ " ... parts left out ..." ],
    "require": {
        "php": ">=5.3.3",
        "symfony/symfony": "2.2.*",
        "doctrine/orm": "~2.2,>=2.2.3",
        "doctrine/doctrine-bundle": "1.2.*",
        "twig/extensions": "1.0.*",
        "symfony/assetic-bundle": "2.1.*",
        "symfony/swiftmailer-bundle": "2.2.*",
        "symfony/monolog-bundle": "2.2.*",
        "sensio/distribution-bundle": "2.2.*",
        "sensio/framework-extra-bundle": "2.2.*",
        "sensio/generator-bundle": "2.2.*",
        "jms/security-extra-bundle": "1.4.*",
        "jms/di-extra-bundle": "1.3.*"
    },

    [ " ... parts left out ..." ],

    "minimum-stability": "alpha",
    "extra": {
        "symfony-app-dir": "app",
        "symfony-web-dir": "web",
        "branch-alias": {
            "dev-master": "2.2-dev"
        }
    }
}
```

Copy the contents of the require key and paste them into your composer.json file, being sure to only replace the core Symfony libraries, and not any custom lines you may have added. In this case, the doctrine-fixtures-bundle is custom, so I'll leave it alone:

```
"require": {
    "php": ">=5.3.3",
    "symfony/symfony": "2.2.*",
    "doctrine/orm": "~2.2,>=2.2.3",
    "doctrine/doctrine-bundle": "1.2.*",
    "twig/extensions": "1.0.*",
    "symfony/assetic-bundle": "2.1.*",
    "symfony/swiftmailer-bundle": "2.2.*",
    "symfony/monolog-bundle": "2.2.*",
    "sensio/distribution-bundle": "2.2.*",
    "sensio/framework-extra-bundle": "2.2.*",
    "sensio/generator-bundle": "2.2.*",
    "jms/security-extra-bundle": "1.4.*",
    "jms/di-extra-bundle": "1.3.*",.

    "doctrine/doctrine-fixtures-bundle": "dev-master"
},
```

Also, be sure you have the `minimum-stability` set to `alpha`. As Stof points out in the blog, this is because at least one of the libraries here is still technically at an alpha state. This allows that library to be included. Finally, add the branch-alias that maps `dev-master` to `2.2-dev`.

Updating with Composer¶

All we need to do now is tell Composer to re-read the `composer.json` file and update everything. But wait! As you undoubtedly remember from watching our wildly entertaining and informative tutorial on Composer, we need to run `composer.phar update` to do this, which *can* be a dangerous command. Let's run `update` now, and then talk about what horrible things this might be doing:

```
php composer.phar update
```

Remember that running `update` will update *all* of our vendor libraries to the latest versions specified in `composer.json`. Since the `doctrine-fixtures-bundle` is tagged at `dev-master`, it means that it is updating this bundle to the latest commits on the `master` branch.

Instead of running a naked `update`, you could try to specify only the libraries you want to update:

```
php composer.phar update symfony doctrine/orm doctrine/doctrine-bundle twig sensio jms
```

But since so many libraries depend on the version of Symfony, you'll quite likely get dependency errors if you try this. Give it a shot, but your best option is to tag as many packages to specific versions as possible before running `update`. If a library you use isn't tagged, well, it's time to give the maintainer a loving poke to tag.

Upgrading your Project¶

Ok! We're now on Symfony 2.2! All we need to do now is see if any of our code needs to be updated. In fact, when I refresh the page, Symfony 2.2 kills my project!

> Cannot import resource "@FrameworkBundle/Resources/config/routing/internal.xml" from "/Users/weaverryan/Sites/knp/casts/new-2.2/app/config/routing.yml". Make sure the "FrameworkBundle" bundle is correctly registered and loaded in the application kernel class.

Ok, don't panic. Head back to the blog post and find 2 UPGRADE links. The first UPGRADE file is a big list of all the backwards-compatibility breaks in Symfony, which may or may not affect you depending on which features you use.

The second UPGRADE file talks about changes that you'll need to make to the files in the Symfony Standard Distribution, which was the starting point of your project. It mentions a change to the `_internal` route used for ESI caching, which sounds just like the error we're seeing.

Chapter 3: Fragments, ESI and Caching

FRAGMENTS, ESI AND CACHING¶

Symfony 2.2 comes with a brand new fragments sub-framework, which allows you to render small parts of your page - like our inner box - independently. Actually, this has existed since Symfony 2.0, but was called “sub-requests”. In 2.2, the feature has been overhauled for flexibility, speed and security.

Understanding Http Caching, ESI and Fragments¶

One of the best features of Symfony is its use of Edge Side Includes or ESI. This is where different parts of your page are rendered as small `esi` tags. Before your user sees these, a middle layer parses them out, makes another request back to your app for just that fragment, and then combines it all together.

This is the “fragments” framework at work: the page is broken down into small pieces and each has a special URL to render it all by itself. This lets you cache the different fragments of your page independently. Since the middle cache layer puts all the fragments together, your user has no idea you’re doing all this voodoo behind the scenes.

Fragments in Symfony2¶

The easiest way to split your page into these fragments is with the `render` tag we’re using, which gets its content by rendering another controller:

```
{# This is the 2.1 syntax for fragments (or sub-requests) %}
{% render 'EventBundle:NewFeatures:inner' with {
    'color': 'lightblue'
}, {
    'standalone': true
} %}
```

There’s a lot more to know about this, so check out Symfony’s Http Caching chapter. The important point is that your page can be broken down into fragments, and even though we don’t have a route that points to a fragment like `innerAction`, there’s a special URL that let’s us render it independently.

Welcome fragments and ProxyListener, goodbye internal.xml¶

Before Symfony 2.2, this special URL came from importing the `internal.xml` routing file, which exposed a regular Symfony route that was capable of rendering any controller.

```
# app/config/routing.yml
_internal:
    resource: "@FrameworkBundle/Resources/config/routing/internal.xml"
    prefix:   /_internal
```

If you used this route, you were supposed to somehow make sure that the URL was protected so that only your caching layer could use it. If a normal user had access to this, they could render any controller with any arguments in your system, which would be a bummer...

In Symfony 2.2, the `internal.xml` routing file is gone. Let’s remove it and replace it with a `fragments` key in `config.yml`. Instead of a route, this activates a listener that watches for any requests that start with `/_proxy`, which is the URL that the ESI tags now render as. This alone doesn’t help security, except that the listener uses a few tricks internally, which we’ll talk more about in a moment.

```
# app/config/config.yml
framework:
    esi:          ~
    fragments:    { path: /_proxy }
    # ...
```

The new Twig render Syntax¶

For now, let's get our application working! Aside from this configuration change, the `render` tag now looks different. First, it isn't a tag at all anymore, it's now a function that's rendered using the double-curly brace syntax:

Tip

The `{% render %}` tag will still be supported until Symfony 3.0.

Second, when you reference the controller, you must wrap it in a call to a new Twig `controller` function. For now, I'll remove the `standalone` key that activated ESI:

```
{{ render(controller('EventBundle:NewFeatures:inner', {
    'color': 'lightblue'
})) }}
```

When we refresh the page, things finally work! Looking at the timeline, we see information about the main request and the `inner` fragment. This is how things look when we're not using ESI - the full page and the `inner` fragment render all at once. This is called the `default` rendering strategy and it's pretty straightforward.

Activating ESI¶

Since that's boring, let's activate ESI! To do this, just change the function to `render_esi`:

```
{{ render_esi(controller('EventBundle:NewFeatures:inner', {
    'color': 'lightblue'
})) }}
```

Refresh again! It renders exactly the same, how exciting! But actually, a lot just changed behind the scenes. The main page now renders everything except the inner area. Instead, it prints out an ESI tag. Our caching layer parses it, makes another request into Symfony for that piece and then combines it all together. This is called the ESI rendering strategy, because the first main request returns an ESI tag in place of the inner area.

Debugging ESI with X-Symfony-Cache¶

We're using Symfony's reverse proxy in PHP, so all of this happens on the server and is completely invisible to us. But if you view the network details, you'll see an `X-Symfony-Cache` header, which describes what's happening at our caching layer. You can now see two entries - one for the main page request and another when the caching layer requests just the inner portion.

> X-Symfony-Cache: GET /new/fragments: miss; GET /_proxy?_path=color%3Dlightblue%26_format=%3Dhtml%26_controller%3DEventBundle%253ANewFeatures%253Ainner: miss

Of course, we're not actually caching either part, but you can see how each operates independently.

Using HInclude Tags¶

To push things further, change the function to `render_hinclude`.

```
{{ render_hinclude(controller('EventBundle:NewFeatures:inner', {
    'color': 'lightblue'
})) }}
```

Refresh your page to see that the inner section has vanished! When you view the source you'll find an HTML tag with a URL. This is called an `hinclude` tag, and it works a lot like an ESI tag. In both cases, an extra request is made back to the server to fetch the content, which allows that small piece to be cached independently. The difference is that this tag is processed by your client using a JavaScript library called HInclude whereas ESI is processed in a layer somewhere inside your server architecture, invisible to the user.

Fragment URL Security¶

Let's look a little bit more at the URL in the HInclude tag. If we open this URL directly we can see the content that will be rendered. In fact, regardless of whether you use the ESI or HInclude strategy, this URL is used to allow an outside layer to request the individual fragments. This was activated by adding the `fragments` key to `config.yml`.

So what prevents an evil user from exploiting this URL to render any controller in our system with any parameters? Nothing! Just kidding, there are two built in protections: *trusted proxies* and *signed URLs*.

Trusted Proxies¶

The class that handles all this magic is called `FragmentListener`. Before it starts serving anything from your application, it first checks to see if the person requesting is "trusted".

If you're using a reverse proxy like Varnish, then you'll want to add its IP address or - CIDR IP address range for the super-geeks - to your `config.yml` file:

```
framework
    trusted_proxies:
        - 192.168.12.0
```

Note

Internally, this sets the `Request::setTrustedProxies` method. Currently it appears that IP ranges (e.g. `192.168.12.0/23`) are respected in `FragmentListener`, but aren't accepted under the `trusted_proxies` key. This was fixed in Symfony 2.3.

If the request comes from this IP or range, it allows it. And, if it comes from a local address, it also allows it. In other words, if it's someone you trust, then it's ok.

Signed URLs¶

If it's not, then it falls back to use URL signing. Notice the `_hash` query parameter at the end of the URL. That's generated using an application secret and the URI. It means that if we weren't trusted, we could still access this exact URL. But if we changed any part of it, it wouldn't match the hash and Symfony would deny access. It's a pretty clever way to expose parts of your application that you want, without exposing everything.

Phew! Let's change back to use the ESI strategy and keep going with some of the other great new features in Symfony 2.2.

Chapter 4: Hostname Routing

HOSTNAME ROUTING¶

A lot of people wanted it, so brand new in Symfony 2.2 is the ability to match route names based on the host parameter. To see it in action, let's duplicate the `fragments` route and make the first one point to a non-existent controller:

```
fragments:
    path: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragmentsFake

fragments2:
    path: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragments
```

Perfect! Since both routes have the same pattern, the first always wins and our application breaks.

Pretend now that this route should only respond to `foo.sf22.l`. To make this happen, add a `host` key under the route:

```
fragments:
    path: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragmentsFake
    host:    foo.sf22.l
```

When we refresh, the application works, which means that the first route is no longer matching since we're not at the foo subdomain. The second route has no `host` key, it matches on any host. We can also switch to the subdomain and see that the first route indeed matches.

USING PARAMETERS IN YOUR ROUTES¶

The only problem is that we've hardcoded the domain name, and it's likely that this domain will be different locally than beta and production.

To fix this, let's leverage a feature that was actually added in Symfony 2.1: the ability to use parameters in routing files. Start by adding a new entry in `parameters.yml` called `base_host`:

```
# app/config/parameters.yml
parameters:
    # ...
    base_host:        localhost
```

Tip

Remember that there's nothing special about this file - you can have a `parameters` key in `config.yml` or any other of your configuration files.

Next, update the route to use the parameter under the `host` key.

```
fragments:
    path: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragmentsFake
    host:    foo.%base_host%
```

When we try it, both subdomains behave exactly as before. Easy!

Chapter 5: New Dialog Goodies: Autocomplete, Progress

NEW DIALOG GOODIES: AUTOCOMPLETE, PROGRESS¶

Now let's turn to something completely different: custom console commands. Creating commands in Symfony has always been easy and powerful and if you're new to it, just check out the cookbook article we have on the topic:

```
namespace Yoda\EventBundle\Command;

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

class PlayCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setName('yo:dawg')
            ->setDescription('For playing')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        /** @var $dialog \Symfony\Component\Console\Helper\DialogHelper */
        $dialog = $this->getHelper('dialog');

        $favoriteThing = $dialog->ask(
            $output,
            'What do you want more of? '
        );

        $output->writeln(sprintf(
            'I heard you liked <comment>%s</comment>, so I put more <comment>%s</comment> in your <comment>%s</com
            $favoriteThing,
            $favoriteThing,
            $favoriteThing
        ));
    }
}
```

Symfony 2.2 added a bunch of really fun new features. Right now, we have a simple command, let's make it more awesome!

Choose Options with the new DialogHelper::select¶

Suppose that we want to make sure that one of a few things is chosen. We're already using the DialogHelper to ask for a thing. Now, let's create an array of our favorite items and use the `select` function:

```
protected function execute(InputInterface $input, OutputInterface $output)
{
    /** @var $dialog \Symfony\Component\Console\Helper\DialogHelper */
    $dialog = $this->getHelper('dialog');

    $favoriteItems = array(
        'Symfony',
        'Ice Cream',
        'Documentation',
    );

    $index = $dialog->select(
        $output,
        'What do you want more of? ',
        $favoriteItems
    );
    $favoriteThing = $favoriteItems[$index];

    $output->writeln(sprintf(
        'I heard you liked <comment>%s</comment>, so I put more <comment>%s</comment> in your <comment>%s</comme
        $favoriteThing,
        $favoriteThing,
        $favoriteThing
    ));
}
```

When we try it, we get more of your favorite thing!

Command-Line Auto-completion¶

Let's keep going. Another function on the dialog helper is `askAndValidate`, which has actually always existed. First, create a simple validation function that makes sure the value is one of our things. Next, use the `askAndValidate` function instead of `select`:

```
$validation = function($thing) use ($favoriteItems) {
    if (!in_array($thing, $favoriteItems)) {
        throw new \InvalidArgumentException(sprintf(
            '"%s" is not one of my favorite things!',
            $thing
        ));
    }

    return $thing;
};

$favoriteThing = $dialog->askAndValidate(
    $output,
    'What do you want more of?',
    $validation,
);
```

When we try it, it's pretty simple - we type anything, and it keeps asking us until we enter a valid value. This is actually a bit worse than using `select`, but stay with me!

Change the `askAndValidate` method slightly - passing `false` and `null` to the default max tries and default value arguments and then finally the array of `$favoriteItems` next:

```
$favoriteThing = $dialog->askAndValidate(
    $output,
    'What do you want more of?',
    $validation,
    false,
    null,
    $favoriteItems
);
```

Try this again. At first, it looks the same. But as soon as we type anything, it starts auto-completing our answer. How cool is that!

Showing a Progress Bar¶

Let's add just one more fancy thing. A lot of times, I write console tasks to handle long-running processes. I normally wouldn't admit this but, I can be a bit impatient, I always want to know how far through the process I am. Normally I set some variables and print out a status message. Now there's a much better way.

First, setup a loop that pauses in the middle randomly - this will be our "fake" long-running process:

```
foreach (str_split($favoriteThing) as $char) {
    usleep(rand(100000, 1000000));
}
```

Next, grab the ProgressHelper by using the `getHelper` function. This is brand new to Symfony 2.2 and it works by showing details about how far along the process is. Start it with, well the `start` function, which takes the number of "steps" as its second argument. If you were looping through 1000 database records, you'll probably set this to 1000:

```
/** @var $progressHelper \Symfony\Component\Console\Helper\ProgressHelper */
$progressHelper = $this->getHelper('progress');
$progressHelper->start($output, strlen($favoriteThing));
foreach (str_split($favoriteThing) as $char) {
    // ...
}
```

Now, on each loop, simply call `advance` to move the progress bar through one step:

```
foreach (str_split($favoriteThing) as $char) {
    usleep(rand(100000, 1000000));

    $progressHelper->advance();
}
```

That's it, let's run this and see what happens! This time, we get a really cool progress bar that shows us exactly where things are.

You can even control how this looks, using a number of different functions on the ProgressHelper class. The easiest is `setFormat`, which let's you choose how "verbose" the progress bar should be. Let's choose `FORMAT_VERBOSE` to see the most details possible:

```
$progressHelper->setFormat(
    \Symfony\Component\Console\Helper\ProgressHelper::FORMAT_VERBOSE
);
```

If you're not used to building your own custom console commands, they're easy and powerful! And even if you're not using Symfony, you can use *just* the Console component to create single-file, standalone command-line applications.

Chapter 6: Upgrade to Symfony 3.0?

UPGRADE TO SYMFONY 3.0?¶

Now that we are upgraded to Symfony 2.2 it's time to start preparing your application to work with 3.0!

On each release of Symfony, some functionality is deprecated and scheduled to be removed entirely later. For the first time, a few things have been deprecated and scheduled to be removed in Symfony 3.0.

To see them, check out the Symfony 3.0 CHANGELOG - through any of the 2.x releases, you can check out this file and find out how to be ahead of the game when it comes to future-compatability.

Routing Method and Scheme Changes¶

There's not a lot in here yet, but there are a few really important things about routing! First, let's update our route to use the `_method` and `_scheme` requirements:

```
fragments:
    pattern: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragmentsMobile
    host:    foo.%base_host%
    requirements:
        _method: GET|POST
        _scheme: https
```

All of this works before Symfony 2.2, and it says that this route should only match if the HTTP method is GET or POST and should force the user to use HTTPs.

As of 2.2, this syntax is deprecated, but won't be removed until 3.0. Let's update our routes to be Symfony3-compatible:

```
fragments:
    pattern: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragmentsMobile
    host:    foo.%base_host%
    methods: [GET, POST]
    schemes: https
```

Pattern to Path Routing Change¶

This will work today, tomorrow, and even for any Symfony3 version. Actually, there's one more change that's *much* more important than this: `pattern` has changed to `path` :

```
fragments:
    path: /fragments
    defaults:
        _controller: EventBundle:NewFeatures:testFragmentsMobile
    host:    foo.%base_host%
    methods: [GET, POST]
    schemes: https
```

This is all just a syntax change, and if you get in the habit of using the new way, you might just save yourself some heartache later when Symfony 3.0 is the greatest thing since sliced bread.

Time to go run and tell your friends that you've upgraded to Symfony 3.0! Ok, you haven't actually of course, but if your

friends aren’t programmers they won’t know what you’re talking about anyways.

Chapter 7: Extras

EXTRAS¶

There have been a lot of other wonderful changes as well, and fortunately, there are a lot of great places to learn about them.

New in Symfony 2.2 Blog Series¶

First, Fabien did a fantastic job with a series of blog posts called What's new in Symfony 2.2, which you can find by looking back at December 2012 and January and February of 2013. One of my favorites is the new Stopwatch component, which let's you add your own events that live in the timeline. Actually, this was possible in Symfony 2.1, but now the code is a component and available for anyone to use.

Tracking Changes: CHANGELOG, UPGRADE¶

For other changes, check out the CHANGELOG and UPGRADE files on the Symfony repository itself. The CHANGELOG contains updates between minor releases, and eventually will get quite long. So pour a pint and treat yourself to an exciting read. And actually, each individual component and bundle also has its own `CHANGELOG`. To see what's new in the HttpFoundation component, look right inside of it.

That's it! Symfony 2.2 is a great new release with more features and minor backwards compatibility breaks. If you're still relatively new to Symfony, check out our Starting in Symfony2 series and then start coding!

See ya next time!