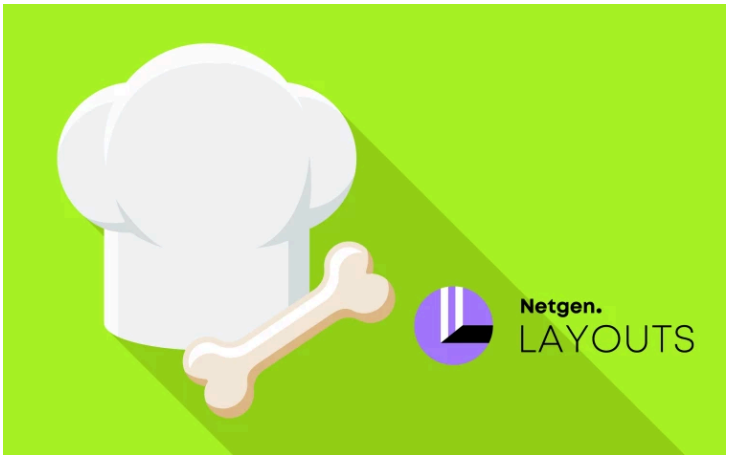
Netgen Layouts: Building Pages with Symfony



Chapter 1: Hello Layouts+ Setup!

Hey friends! I'm *so* glad you're here with me, because this tutorial is about something fun, cool, and *quite* powerful. No, it's not about a masked, crime-fighting feline with superpowers, though that *would* be pretty cool. This tutorial is all about the Netgen Layouts package.

What Is Layouts?

This library has existed for *years*, but I've only recently checked it out. Layouts is, very simply, a way to take any existing Symfony app and add the ability to dynamically rearrange how your pages are organized *on the fly* via an admin section... including adding new dynamic content. It's a really cool mixture of a normal Symfony app with controllers and Twig templates... plus content management features that you can opt into on a page-by-page basis. I *particularly* love the opt-in approach

Who Needs Layouts?

Why would you go to the trouble of using Layouts in your Symfony app? Well, not *all* projects need this. But if an admin user needs to be able to make changes to how your site and its content are organized, then this is for you. This includes being able to add and change collections of items - like featured products - right to the middle of an existing page, rearranging content from a Twig template higher or lower on the page, adding some completely *new* customizable content to a page *or* creating temporary landing pages and allowing all of this to be done by regular ol' users. You can leverage Layouts for a single page on your site, leaving everything else to be a normal Symfony app, or *every* page on your site can use it. Like I said, you *opt into* Layouts as you see fit.

Project Setup

I could go on and on, but it's probably best to see the Layouts magic in action. It's *super* fun to play with, so you should definitely download the course code from this page and code along with me. When you unzip the file, you'll find a `start/` directory with the same code that you see

here. Pop open this `README.md` file for all those sweet setup details. I've *already* gone over to my terminal, installed my Node assets via:

```
yarn install
```

and ran:

```
yarn watch
```

to build my CSS and JS files. But that's all just to make our app and this tutorial more realistic. Layouts doesn't require us to use Encore and it doesn't mess with our CSS and JS at all.

Anyway, the last step in the `README` is to open another terminal tab and run:

```
symfony serve -d
```

to start a web server at https://127.0.0.1:8000 - I'll cheat and click that. And... hello new side project: it's Bark & Bake! Listen, dogs are pretty tired of our lazy attempts at canine cuisine. Crunchy kibble? *No thanks*. So we've built this site to inspire people to be the *best* chefs they can be... for their dogs.

This is a pretty traditional Symfony app with a few controllers and some Twig templates. It also has two entities: A `User` entity for security, and a `Recipe` entity. On the site, we have a homepage that shows the latest and greatest recipes, a recipe section, and the ability to *open* a specific recipe, so we can follow along in the kitchen. That's *it*. This skills stuff isn't implemented at *all* yet.

So, other than being able to edit the details of each recipe via an admin area, this is a static site! Time to change that! Soon, we'll be able to take this homepage - which is written via a normal Symfony controller and template... as you can see here - and use layouts to *dynamically* insert content and rearrange things!

Installing Layouts

So let's get Layouts installed. Find your terminal and run:

```
composer require netgen/layouts-standard
```

This will download several packages, including a couple of new bundles. When it finishes, run:

```
git status
```

to see that it *also* gave us two new route files, which add some admin routes that we're going to see in a few minutes.

Running the Migrations

Layouts also requires some extra database tables where it stores info about the layout's we'll create as well as any custom content we're putting into them. We'll see all of that in the Layouts admin section in a minute. To add the needed tables, scroll up and copy this nifty `doctrine:migrations:migrate` line.

This is kind of cool. The layouts packages *comes* with migrations... and this *executes* those. Paste this command, but if you're using the Docker database setup that we described in the README - I am - then modify this to start with `symfony console` so that it can inject the Docker environment variables that point to our database:

```
symfony console doctrine:migrations:migrate --configuration=vendor/netgen/layouts-core/
```

And... perfect! One caveat is that these migrations are written for MySQL *specifically*. Layouts *only* supports MySQL right now.

Ignoring the Custom Tables

For the most part, *Layouts* is going to entirely manage all of the tables that we just added: we don't need to do anything with them. But now that those exist in our database, if we were to add

a new entity and then generate a migration for that... the migration would try to *drop* all of the Netgen Layouts tables. Watch! Run:

```
symfony console doctrine:schema:update --dump-sql --complete
```

This mimics generating a migration, and... yup! It wants to drops all of the Layouts tables. To fix this, head into `config/packages/doctrine.yaml` and, under `dbal`, add `schema_filter`, and pass a regular expression... which you can copy from the Layouts documentation:

config/packages/doctrine.yaml

```
doctrine:
    dbal:
// ... lines 3 - 7
        schema_filter: ~^(?!nglayouts_)~
// ... lines 9 - 44
```

Perfect! With that, if we go back and run the `doctrine:schema:update` command *again*...

```
symfony console doctrine:schema:update --dump-sql --complete
```

It's *clean*, except for `doctrine_migration_versions`. But, no worries: the `make:migration` command is smart enough not to drop its *own* table.

Ok, Netgen Layouts is installed and it has the database tables it needs. If we go back and refresh our site now... *congratulations*! Absolutely nothing is different. Though, we *do* have a cute little web debug toolbar icon down here that we'll talk about later.

This, again, is one of the great things about Layouts. Just installing it does *not* take over your app. Layouts is not being used at *all* to render this page.

Next, let's dive into the Layouts admin area to create our first layout. Then, we'll see how that interacts with the *real* pages on our site.

Chapter 2: Creating & Mapping Layouts

Ok, let's see what Layouts is all about. In this chapter, we'll, step-by-step, create & use a "layout", learning *exactly* how Layouts works its magic along the way.

To check out the Layouts admin section, head to `/nglayouts/admin` to find... a login form! The login form has nothing to do with the Layouts... it's just that the layouts admin area *requires* you to be logged in... and I've already added a login form to our site. There's even a user in the database! Log in with `doggo@barkbite.com`, password `woof`.

The Security Role Needed for the Admin Area

And when we submit... access denied! No worries: click down on the web debug toolbar's security icon... and go to "Access Decision". Yup: we were denied access because it was looking for a role called `ROLE_NGLAYOUTS_ADMIN`. To access the layouts admin area, we need to have this role.

The simplest way to add it is to go to `config/packages/security.yaml`. The user we're logged in as right now has `ROLE_ADMIN`. So, under `role_hierarchy` *also* give our user `ROLE_NGLAYOUTS_ADMIN`:

config/packages/security.yaml

```
1  security:
↕  // ... lines 2 - 6
7      role_hierarchy:
8          ROLE_ADMIN: [ROLE_USER, ROLE_NGLAYOUTS_ADMIN]
↕  // ... lines 9 - 56
```

Creating our First Layout

And now if we click back, ta-da! Welcome to the layouts admin section! To understand what layouts does... it's best to see it in action. Start in this Layouts section... and click to create a new layout. This shows us about six different layout *types* we can choose from. As you'll see, these are much less important than they might seem at first, because, once you're in a layout,

you can really do whatever you want, including floating things left and right. I typically choose "Layout 2". Call this "Homepage Layout" because I'm planning to use this on our homepage.

And... welcome to the layout editor! Quick tour: these items on the left side are called "blocks", and there are many different types, from simple title blocks to Google maps... to more complex things like lists and grids where you can render dynamic collections of things, like featured recipes. The main things we "do" on this page is choose a block on the left... then drag it onto one of the "zones" in the middle.

Putting Blocks onto the Layout

Grab a "Title" block and drag it somewhere onto the page... then give it some text. Cool!

It's a modest start, but, good enough! In the upper right, hit "Publish Layout".

And now that we have this new layout, open a second tab and go to the homepage to discover that... absolutely nothing changed! Let me actually rearrange my tabs.

Mapping a Layout

Anyways, nothing changed because, once you have a layout, you need to *map* it to a specific page or set of pages. *That's* the job of the layout mapping section. These are really the only two important sections in the admin area.

Here, add a new mapping and then go to Details. There are multiple ways that you can map a layout to a specific URL. You could use, for example, the path info, which is a fancy term that means "the URL, but without query parameters". Or you could use a path info prefix - like use this layout for all URLs that start with "/products". *Or* you can even map a layout to a specific route name.

Let's try that one. Hit "Add target". Then... let's go find our homepage route name: `src/Controller/MainController.php`. Here it is: `app_homepage`:

src/Controller/MainController.php

```
// ... lines 1 - 9
class MainController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(RecipeRepository $recipeRepository): Response
    {
// ... lines 15 - 22
    }
}
```

Move back over, paste and hit "Save target".

We're going to talk about other ways to map or "activate" a layout for pages later. But route and path info are the simplest and flexible. They say:

> *"If the current route or URL matches what we have here, use this layout."*

Hit save changes. To choose *which* layout goes with this mapping, hit "Link layout" and select the only one: "Homepage Layout".

Awesome! So *now* when we go to the homepage, it *should* use the homepage layout. But... what does that even *mean*? Let's find out! Refresh and... we *still* don't see any difference! It's the same static page from Symfony!

Extending the Dynamic Base Layout

Oh, that's because we missed an important installation step. My bad! Go open the template for this page: `templates/main/homepage.html.twig`. Right now, we're extending `base.html.twig`:

templates/main/homepage.html.twig

```
{% extends 'base.html.twig' %}

// ... lines 3 - 60
```

And *that* template, like usual, has a block called `body` in the middle:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 16
    <body>
// ... lines 18 - 46
        {% block body %}{% endblock %}
// ... lines 48 - 60
    </body>
</html>
```

So it's a *super* traditional setup.

Now, change the `extends` to a dynamic variable called `nglayouts.layoutTemplate`:

templates/main/homepage.html.twig

```
{% extends nglayouts.layoutTemplate %}
// ... lines 2 - 60
```

Configuring the Base Layout

Try the page again. Error! That's progress! It says:

> *"Base page layout, not specified. To render the page with Layouts, specify the base page layout with this config."*

This will all make more sense in a minute. What it wants us to do is open `config/packages/` and create a new file - which can be called anything - but let's call it `netgen_layouts.yaml`. Inside, add `netgen_layouts` and, below that, `pagelayout` set to our `base.html.twig`:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
    pagelayout: 'base.html.twig'
```

I'll explain this all in a minute. If we refresh now... huh, same error! It's possible Symfony didn't see my new config file... so let me clear the cache to be sure:

```
php ./bin/console cache:clear
```

And now... yes! It works! Except... it's *still* the same static page! *But*, for the first time, down on the web debug toolbar, it shows that the "Homepage Layout" is being used. So it *realized* the layout should be used... it just doesn't seem to be *rendering* it.

Rendering the layout Block

To fix that, we need to do *one* last thing... then we'll back up and explain what's going on and how cool it is. In `base.html.twig`, around `{% block body %}`, add `{% block layout %}`... then after `{% endblock %}`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 16
    <body>
// ... lines 18 - 46
        {% block layout %}
        {% block body %}{% endblock %}
        {% endblock %}
// ... lines 50 - 62
    </body>
</html>
```

Refresh one more time. And... whoa! Our page is gone! Okay, we still have the nav and footer... which come from above and below the blocks in `base.html.twig`, but the actual *contents* of our page are gone and replaced by the dynamic title block! What Black Magic is this?

The Layouts Template Inheritance Magic

First, before I explain, let me say that there are *much* faster ways to start with Netgen Layouts: they have starter projects for normal Symfony apps, Sylius apps and Ibexa CMS apps. But we did all this set up work *manually* on purpose... because I *really* want you to understand *how* Layouts works: it's surprisingly simple.

First, our page is still hitting our normal route - `app_homepage` - and it's *still* executing our normal controller and *still* rendering our normal template. No magic there at *all*.

But then, we extend `nglayouts.layoutTemplate`. What does *that* point to? If there is *no* layout mapped to a particular page, `nglayouts.layoutTemplate` will resolve to

`base.html.twig`. That's thanks to the config we added here:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
2      pagelayout: 'base.html.twig'
```

But if layouts *does* find a layout mapping for this page, then `nglayouts.layoutTemplate` resolves to a core Layouts template. In this case, if you hit `Shift`+`Shift`, it's called `layout2.html.twig`... since we selected "Layout 2".

This renders the dynamic layout via these `nglayouts_render_zone` tags: each of these refers to a different section - or "zone" - inside our layout.

Anyways, what's *really* important is that it renders the layout into a Twig block called `layout`. It then extends `ngLayouts.pageLayoutTemplate`, which resolves to *our* `base.html.twig`.

The end result is that our page renders *completely* normally and it still extends `base.html.twig`... but a block called `layout` has been *added* that holds the contents of the dynamic layout.

That's why we didn't see any changes on the page at first. Until we actually *included* `{% block layout %}` in `base.html.twig`, the layout *was* loading... we just weren't *rendering* it anywhere.

The takeaway is this: if you're on a page that does *not* map to a layout, everything is *exactly* the same as always. But if you *are* on a page that maps to a layout, it simply means that you now have a block called `layout` whose contents are equal to whatever you have *inside* of that layout.

Extending the Dynamic Layout on All Pages

So as I mentioned earlier, we don't have to add layouts to every page on our site: we could add it to the homepage and be done! But every page that we want to *support* layouts needs to extend `nglayouts.layoutTemplate`. The nice thing is, even if we extend this, nothing *happens* unless we actually *map* a layout to this page. So, there's no downside to using it everywhere. I'll quickly update `login.html.twig` to use it:

templates/security/login.html.twig

```
{% extends nglayouts.layoutTemplate %}
// ... lines 2 - 39
```

then `list.html.twig` and `show.html.twig`:

templates/recipes/list.html.twig

```
{% extends nglayouts.layoutTemplate %}
// ... lines 2 - 33
```

templates/recipes/show.html.twig

```
{% extends nglayouts.layoutTemplate %}
// ... lines 2 - 38
```

I can really move fast when I need to!

Back in the browser, the recipe list and recipe show pages still look the same... because no layout is resolved. But they're now *ready* to use layouts, if we want to.

Now, as interesting as it is to dynamically control the content on the homepage, we uh, kind of did too much! All of our old content is gone. Is it possible to mix dynamic content with some of the static content from our homepage Twig template? Absolutely. And that's a big part of what makes layouts special. That's next.

Chapter 3: Adding Twig Blocks to your Dynamic Layout

So we just *completely* replaced our homepage with a dynamic layout. But, that's not really *that* interesting. What I *really* want to be able to do is use my *existing* homepage template and all this good content I've prepared:

templates/main/homepage.html.twig

```
{% extends nglayouts.layoutTemplate %}

{% block body %}
    <div class="hero-wrapper">
        <h1 class="header">Bark & Bake</h1>
        <p class="text-center">Doggone Good Treat & Meal Recipes</p>
        <div class="d-flex justify-content-center">
            <img src="{{ asset('images/dog-bone.png') }}" width="auto"
height="50" alt="dog bone icon">
        </div>
    </div>
// ... lines 11 - 58
{% endblock %}
```

but then *tweak* it by adding little bits of dynamic content here and there... or even rearrange things. To do that in the layout, under the blocks, at the bottom, add a special one called "Twig Block"... and let's put that right below the title. Notice that you can put as many blocks as you want inside of a single zone. These zones don't really end up being *all* that important.

Anyways, when you click a block, on the right side, you'll see that block's options. This has an important one called "Twig block name". Enter `body` to match the `{% block body %}` that we have in the template:

templates/main/homepage.html.twig

```
{% extends nglayouts.layoutTemplate %}

{% block body %}
// ... lines 4 - 58
{% endblock %}
```

Ok, hit "publish and continue editing"... then go over and refresh the homepage. Holy content batman! Our Twig content now lives *inside* this dynamic page. That's awesome! And everything *still* works: even the fancy "live component" in the center of the page.

Adding More Blocks to your Template

Okay, so this is cool... but it's still just a bunch of dynamic content on top... then Twig template content on the bottom: we can't *really* mix anything into the *middle* of our page.

Unless... we add more *blocks* to our template. For example, keep the `block body`... just so the page keeps working even if we *don't* map a layout... but then add a `{% block hero %}` around the top section, a block called, how about, `latest_recipes`, `{% endblock %}`, another called `subscribe_newsletter`, `{% endblock %}` and a final one called `featured_skills`, `{% endblock %}`:

templates/main/homepage.html.twig

```
// ... lines 1 - 2
{% block body %}

{% block hero %}
    <div class="hero-wrapper">
// ... lines 7 - 11
    </div>
{% endblock %}

{% block latest_recipes %}
    <div class="container">
// ... lines 17 - 31
    </div>
{% endblock %}

{% block subscribe_newsletter %}
    <div class="text-center pt-4 pb-5 my-4" style="background-color: #fdedf0;">
// ... lines 37 - 40
    </div>
{% endblock %}

{% block featured_skills %}
    <div class="container py-4 my-5">
// ... lines 46 - 65
    </div>
{% endblock %}

{% endblock %}
```

If we stopped now, this would make *no* difference to our app: we're still rendering the `body` block down here... which includes all of those. But we just gave ourselves a *lot* of new power.

Check it out: change the `body` block name to `hero`. And then let's add a few more Twig blocks. Render `latest_recipes` for this one. Oh, by the way, the block "labels" are just for us in the admin area: just for clarity. If I enter "Latest Recipes", that shows up above the block. Totally optional.

Add two more blocks: one that renders `subscribe_newsletter` and finally one for `featured_skills`. Then, up here, I'm going to remove the `title` block for now.

By the way, I'm using the word "block" to mean two different things at once. Blocks are the "things" we add to our layout - like a title, Google map, or list of items. But blocks *also* refer to

the Twig blocks in our templates. And of course, one of the *types* of blocks we can add... is one that renders... Twig blocks. A little confusing - but that's as bad as it gets.

Anyways, say "Publish and continue editing"... then go refresh the frontend. And... sweet! Our page works. I know, it looks exactly like it did a minute ago, but it's now being rendered by layouts... *and* we can rearrange the pieces!

Watch: I'll move the `subscribe_newsletter` down to the bottom, hit "Publish and continue editing", refresh, and... boom! That static part of the page magically moved to the bottom. That is *cool*.

Or, we could move that back up... then add some dynamic content, like text, in between one of the other blocks.

Next, let's get even more aggressive and flexible by allowing the top navigation and bottom footer to be optional, but easy to add, inside the Layout.

Chapter 4: Shared Layouts

Open up `base.html.twig` and move the `{% block layout %}` to be around *everything*. So, put the start just inside the `body` tag... and the end just before the *closing* `body` tag:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 16
    <body>
    {% block layout %}
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
// ... lines 20 - 45
        </nav>

        {% block body %}{% endblock %}

        <div class="container mt-5">
// ... lines 51 - 60
        </div>
    {% endblock %}
    </body>
</html>
```

If we refresh the homepage now... it's destroyed! The top `nav` and `footer` are gone. Why did I do this? Because I love chaos! Kidding - I did it because it gives us the power, inside layouts, to design *totally* custom pages: even pages *without* the traditional `navigation` and `footer`, maybe like a temporary landing page for a promotion.

But let's be honest, 99% of the time, we *will* want the `nav` and `footer`. No problem, head back over to `base.html.twig`. Remember: adding blocks give us more flexibility. So, above the navigation, add a new block called `navigation`, with `{% endblock %}` after. Then, down here, another called `footer`... and `{% endblock %}`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 16
    <body>
    {% block layout %}
        {% block navigation %}
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
// ... lines 21 - 46
        </nav>
        {% endblock %}

        {% block body %}{% endblock %}

        {% block footer %}
        <div class="container mt-5">
// ... lines 54 - 63
        </div>
        {% endblock %}
    {% endblock %}
    </body>
</html>
```

I bet you know what I'll do next. In the layout admin, we can *now* add a Twig block to the top that renders `navigation`... then one down here on the bottom. It doesn't need to be in this last zone... but it makes sense there. Render `footer`.

Let's try it! Hit "Publish and continue editing" and... refresh. We are back!

Creating a Second Layout

Let's create a *second* layout, this time for the `/recipes` page. If you look at `RecipeController`, you'll see that I already did all the work to query for the recipes, and pass them into this template:

src/Controller/RecipeController.php

```
// ... lines 1 - 12
class RecipeController extends AbstractController
{
    #[Route('/recipes/{page<\d+>}', name: 'app_recipes')]
    public function recipes(RecipeRepository $recipeRepository, int $page = 1): Response
    {
        $queryBuilder = $recipeRepository->createQueryBuilderOrderedByNewest();
        $adapter = new QueryAdapter($queryBuilder);
        /** @var Recipe[]|Pagerfanta $pagerfanta */
        $pagerfanta = Pagerfanta::createForCurrentPageWithMaxPerPage($adapter, $page, 4);

        return $this->render('recipes/list.html.twig', [
            'pager' => $pagerfanta,
        ]);
    }
// ... lines 27 - 34
}
```

And *in* that template, we loop over and render each one, *with* pagination:

templates/recipes/list.html.twig

```
// ... lines 1 - 4
{% block body %}
    <div class="hero-wrapper">
        <h1>Doggone Good Recipes</h1>
        <p>Recipes your pup will love!</p>
    </div>
// ... lines 10 - 31
{% endblock %}
```

And so, I *definitely* want to include all of this custom work in the new layout.

Back in the admin area, I'll hit "Publish layout" as an easy way to get back to the layout list. Then hit new layout, I'll choose my favorite layout 2 and call it "Recipes List Layout". To start, add a new block called "Full View"... and drag it anywhere onto the page, whoops! There we go.

What *is* this "Full View". It's nothing special, in fact, it's kind of redundant! It's nothing more than a "Twig block" that renders the block called `body`. So, yes, we could have *just* as easily done this by *using* the normal Twig block and typing in `body`.

Publish this layout... then go to "Layout Mappings". Add a new one... and this time I'll link it first... to "Recipes List Layout". Then click "Details". Like last time, we could map this via the

route name. But to see something different, use "Path Info", which, again, is just a fancy word for the URL, but without any query parameters. Make it match `/recipes`... "Save Changes" and... sweet!

When we try the page... it looks awesome! Except, whoops, I forgot the nav and footer! Adding those two blocks to "Recipe List Layout" is easy. But what if, later we decide that every page should render both the navigation block on top as *well* as a dynamic banner, maybe for a sale that we're having. If that happened, we would need to edit *every* layout to manually add that new banner.

Shared Layouts

Fortunately, there's a better way to handle repeated layout elements like this.

Hit "Discard" to get back to the layouts list, then click "Shared layouts" and "New shared layout". As usual, the layout type doesn't matter much, so I'll use my normal one... and call it "Nav & Footer Layout".

This is not going to be a *real* layout that's linked to any pages. Nope, it's just going to be a layout that we steal pieces from. Up in the top zone, create a Twig Block that renders `navigation`... and I'll even label it "Top Nav" to make it more clear. Then in any *other* zone - you can put it at the bottom, but you don't have to, add another twig block that renders `footer` and is *labeled* Footer.

Cool! Hit "Publish layout". Now we have *one* shared layout. Again, these are *not* meant to be mapped to pages: they're meant for us to *use* in other *real* layouts.

Check it out: edit "Recipe List Layout". On the bottom left of the screen, hiding behind the web debug toolbar - I'll close that temporarily - there's a button to link a zone with a shared layout zone. Click that, then select the *top* zone... called the "Header" zone, though that name isn't important.

Now, we can find a shared zone from a shared layout... and we only have one. Hit "Select Zone" and... that's it! The top zone in our layout will now equal whatever block or blocks are in the top zone of that *shared* layout. If we added more stuff to that zone in the shared layout, it would automatically show up here.

Do that one more time: select the last zone so that the footer definitely shows up at the bottom, select the shared zone and... done!

Publish that, move over, refresh and... the full page is back! Let's quickly repeat that for the "Homepage Layout". Oh, but this is tricky because I put *all* of my blocks inside this top zone. Mostly, these zones don't matter, but in this case, to avoid overwriting *all* of this, I'll drag everything except for the navigation twig block down here. We can fix the order later.

And now, set the top zone to use the one from the shared layout. Yup! It replaced what we had there before. Below, also link the *bottom* zone with the shared layout.

Perfect! Let's check the order of our blocks... which is kind of the beauty of layouts. If I don't like the order of what's on my page, I can *always* change it! That's better. Publish the layout, head back to the homepage on the frontend and... beautiful!

Next: let's make our recipe list page more flexible by allowing this top `h1` area to be built and customized from inside layouts... instead of it being hardcoded in the template.

Chapter 5: Adding More Customized Blocks

We're going to work more on this Recipe List Layout later. But, let's do one more things right now. Edit that layout. I want to give our admin users the flexibility to change this *title*. Cool! Let's add a new title block right above... and enter some text.

Hit "Publish and continue editing"... then go to the frontend. What I'm *attempting* to do is replicate this title, or "hero" area - so that we can *remove* it from our Twig template. But when we refresh, that doesn't look right yet.

Go over and look at that template. Ok: to replicate this, we need an `h1` tag wrapped in a `hero-wrapper` div:

templates/recipes/list.html.twig

```
// ... lines 1 - 4
{% block body %}
    <div class="hero-wrapper">
        <h1>Doggone Good Recipes</h1>
        <p>Recipes your pup will love!</p>
    </div>
// ... lines 10 - 31
{% endblock %}
```

Right now, layouts is simply rendering an `h1`. And, by the way, you can, in the title block options, choose between `h1`, `h2`, or `h3`. `h1` is what we need this time.

Adding a Wrapper Div Column

So: how can we wrap this in a `div` and give it a `hero-wrapper` class? The answer: add a nifty "column" block... then move the title *into* that column. Cool right? Finally, when you click on the column, you can add any class you want. Add `hero-wrapper`.

Let's try it! Hit "Publish and continue editing", refresh the frontend and... much better! What about that text? Copy it, add a new "text" block right below our "title" and... paste. Publish and continue editing again... try the frontend again and... look at that! A perfect replica!

To celebrate, over in the template, we can remove that section entirely:

templates/recipes/list.html.twig

```
// ... lines 1 - 5
    <div class="hero-wrapper">
        <h1>Doggone Good Recipes</h1>
        <p>Recipes your pup will love!</p>
    </div>
// ... lines 10 - 33
```

The end result is the *same* as before... except admin users *now* have the ability to change the text.

Custom CSS in Layouts or Pre-Made Custom Block Type?

Though, you probably noticed that this *did* require me to be a bit technical: I had to know the CSS class that the column needed. If the admin users designing your layouts *are* a bit technical, then this might be no problem. But if your editors are *less* technical, you could, instead, create a custom block type - like a hero block - where the user just types in the text and you render this whole thing *for* them. We're not going to create custom blocks in this tutorial... but that's mostly because, by the end of the tutorial, you'll know everything you need to follow the docs for that.

The Layouts Web Debug Toolbar

All right, back on the front end, layouts comes with its own web debug toolbar icon. And if you click this, it's pretty cool. We're going to use this a *bunch* of times. It shows you the layout that was resolved and even the *reason* why it was chosen.

But the *really* useful thing is the "Rendered blocks" section. This shows us all the layouts *blocks* that were rendered to build this page. You can see there's one called "Twig block" for the top nav, a "Column", then the "Title", "Text", "Full view" block and finally the last "Twig" block for the footer. This is a great way to see all the different blocks that are being rendered, as well as the *template* behind each one. Later, we're going to talk about overriding those templates, so we can customize how they look.

Linking to the Layouts Admin

Back in the Layouts admin, publish the layout to get back to the main page. If you go to `/admin`, you'll find that our app already has EasyAdmin installed. Let's add a link from the menu here to Layouts to make life easier. Open `src/Controller/Admin/DashboardController.php`... and find `configureMenuItems()`. Add another with `yield MenuItem::linkToUrl()`, call it "Layouts" and give it some icons: `fas fa-list`. For the url, use `this->generateUrl()` and pass in the route name, which happens to be `nglayouts_admin_layouts_index`:

src/Controller/Admin/DashboardController.php

```
// ... lines 1 - 12
class DashboardController extends AbstractDashboardController
{
// ... lines 15 - 34
    public function configureMenuItems(): iterable
    {
// ... lines 37 - 38
        yield MenuItem::linkToUrl('Layouts', 'fas fa-list', $this->generateUrl('nglayouts_admin_layouts_index'));
    }
}
```

Perfect! That's a small detail, but now when we're on `/admin`, we can click "Layouts" to jump right there.

Okay, status check! We can render Twig blocks and mix in title, text, HTML, Google Maps and other blocks wherever we want. The more Twig blocks we have in the template, the more flexibility we have here.

But what about being able to render a collection of recipes from our database, like the "Latest Recipes" we have on the homepage? That's a big piece of layouts: so let's start diving into it next.

Chapter 6: Adding Lists: Value Type

We have a `Recipe` entity and, on the frontend, a page that lists the recipes. We also saw how easy it is to create a layout, which instantly makes parts of this page configurable.

Adding Lists of Existing Content via Layouts?

But now, looking at the homepage, I'm wondering if we can add more complex blocks, beyond just text. Could we, for example, add a block that renders a list of recipes? Something similar to what we have here right now... except instead of adding this via a Twig block, it's added entirely via layouts by an admin user? And, to go further, could we even let the admin user *choose* which recipes to show here?

Totally! If the *first* big idea of Layouts is allowing Twig template blocks to be rearranged and mixed with dynamic content, then the *second* big idea is allowing pieces of *existing* content - like recipes from our database - to be embedded onto our page by admin users.

How? Edit the Homepage Layout. In the blocks on the left, check out this one called "Grid". Add that after our "Hero" Twig block. The Grid allows us to add individual *items* to it... which could be *anything*. But, I don't see a way to do that!

Ok, so we know that a lot of blocks, like titles, maps, markdown, etc can be added to our pages in layouts out-of-the-box with no extra setup work. But the purpose of *some* blocks - like List, Grid, and the Gallery blocks down here (which are just fancy grids that have JavaScript behavior attached to them) - is to render a collection of "items" that are loaded from *somewhere else*, like our local database, CMS, or even your Sylius store. The "things" or "items" we can add to these blocks are called "value types". And... we currently have *zero*. If this were a Sylius project, we could install the Sylius and Layouts integration and instantly be able to select products. The same is true if you're using Ibexa CMS.

Adding a Value Type

So here's our next big goal: to add our `Recipe` Doctrine entity as a "value type" in layouts so that we can create lists and grids containing recipes.

Step one to adding a value type is to tell Layouts about it in a config file. Over in `config/packages/netgen_layouts.yaml`, very simply, say `value_types`, and below that, `doctrine_recipe` . This is the *internal name* of the value type, and we'll refer to it in a few places. Give it a human-friendly `name` - `Recipe` - and for now, set `manual_items` to `false`... and make sure that has an "s" on the end:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 3
4      value_types:
5          doctrine_recipe:
6              name: Recipe
7              manual_items: false
```

We'll talk about `manual_items` more later, but it's easier to set this to `false` to start.

Head over, refresh our layouts page (it's okay to reload it)... and check out our Grid block! There's a new "Collection type" field and "Manual collection" is our only option right now. So... this still doesn't seem to be working. I can't change this to anything else... and I also can't select items manually.

Dynamic vs Manual Queries

There are *two* ways to add items to one of these "collection" blocks. The first is a *dynamic* collection where we choose from a pre-made query. We could choose a "Most Popular" query that would query for the most popular recipes or a "latest recipes" query, to give two examples. The *second* way to choose items is *manually*: the admin user literally selects which they want from a list.

Adding a Query Type

We're going to start with the first type: the *dynamic* collection. We don't see "Dynamic collection" as an option yet because we need to create one of those pre-made queries first. Those pre-made queries are called `query_types`. We could, for example, create a query type for `Recipe` called "Most Popular" and another one called "Latest".

How do we create those? Head back to the config file, add `query_types` and below that, let's say `latest_recipes`. Once again, this is just an "internal name". Also give it a human-readable `name`: `Latest Recipes`:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 8
9      query_types:
10         latest_recipes:
11             name: 'Latest Recipes'
```

So... what do we do now? If we head back and refresh... we get a very nice error that *tells* us what to do next:

> *"Query type handler for `latest_recipes` query type does not exist."*

It's trying to tell us that we need to build a *class* that represent this query type! Let's do it!

The Query Type Handler Class

Over in the `src/` directory, I'm going to create a new `Layouts/` directory: we'll organize a lot of our custom Layouts stuff inside here. Then add a new PHP class called... how about `LatestRecipeQueryTypeHandler`. Make this implement `QueryTypeHandlerInterface`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
↕  // ... lines 1 - 2
3  namespace App\Layouts;
↕  // ... lines 4 - 5
6  use Netgen\Layouts\Collection\QueryType\QueryTypeHandlerInterface;
↕  // ... lines 7 - 8
9  class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
10 {
↕  // ... lines 11 - 29
30 }
```

Then go to "Code Generate" (or `Command`+`N` on a Mac), and select "Implement methods" to add the four we need:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 4
use Netgen\Layouts\API\Values\Collection\Query;
// ... line 6
use Netgen\Layouts\Parameters\ParameterBuilderInterface;

class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
    public function buildParameters(ParameterBuilderInterface $builder): void
    {
        // TODO: Implement buildParameters() method.
    }

    public function getValues(Query $query, int $offset = 0, ?int $limit = null): iterable
    {
        // TODO: Implement getValues() method.
    }

    public function getCount(Query $query): int
    {
        // TODO: Implement getCount() method.
    }

    public function isContextual(Query $query): bool
    {
        // TODO: Implement isContextual() method.
    }
}
```

Nice! Let's see... I'll leave `buildParameters()` empty for a minute, but we'll come back to it soon:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 9
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 12 - 15
    public function buildParameters(ParameterBuilderInterface $builder): void
    {
    }
// ... lines 19 - 40
}
```

The most important method is `getValues()`. This is where we'll load and return the "items". If our recipes were stored on an API, we would make an API request here to fetch those. But

since they're in our local database, we'll query for them.

To do that, go to the top of the class, add a `__construct()` method with `private RecipeRepository $recipeRepository`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 4
use App\Repository\RecipeRepository;
// ... lines 6 - 9
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
    public function __construct(private RecipeRepository $recipeRepository)
    {
    }
// ... lines 15 - 40
}
```

Then, down in `getValues()`, `return $this->recipeRepository`... and use a method that I already created inside of `RecipeRepository` called `->createQueryBuilderOrderedByNewest()`. Also add `->setFirstResult($offset)` and `->setMaxResults($limit)`. The admin user will be able to choose *how* many items to show and they can even *skip* some. And so, Layouts passes us those values as `$limit` and `$offset`... and we use them in our query. Finish with `->getQuery()` and `->getResult()`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 9
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 12 - 19
    public function getValues(Query $query, int $offset = 0, ?int $limit = null): iterable
    {
        return $this->recipeRepository->createQueryBuilderOrderedByNewest()
            ->setFirstResult($offset)
            ->setMaxResults($limit)
            ->getQuery()
            ->getResult();
    }
// ... lines 28 - 40
}
```

Perfect! Below, for `getCount()`, let's do the exact same thing... except we don't need `->setMaxResults()` or `->setFirstResult()`. Instead, add

`->select('COUNT(recipe.id)')`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
↕ // ... lines 1 - 9
10 class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
11 {
↕ // ... lines 12 - 28
29     public function getCount(Query $query): int
30     {
31         return $this->recipeRepository->createQueryBuilderOrderedByNewest()
32             ->select('COUNT(recipe.id)')
33             ->getQuery()
↕ // ... line 34
35     }
↕ // ... lines 36 - 40
41 }
```

I'm using `recipe` because, over in `RecipeRepository`... if we look at the custom method, it uses `recipe` as the alias in the query:

src/Repository/RecipeRepository.php

```
↕ // ... lines 1 - 17
18 class RecipeRepository extends ServiceEntityRepository
19 {
↕ // ... lines 20 - 42
43     public function createQueryBuilderOrderedByNewest(string $search = null):
   QueryBuilder
44     {
45         $queryBuilder = $this->createQueryBuilder('recipe')
↕ // ... lines 46 - 53
54     }
55 }
```

After that, update `->getResult()` to be `->getSingleScalarResult()`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 9
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 12 - 28
    public function getCount(Query $query): int
    {
        return $this->recipeRepository->createQueryBuilderOrderedByNewest()
            ->select('COUNT(recipe.id)')
            ->getQuery()
            ->getSingleScalarResult();
    }
// ... lines 36 - 40
}
```

Phew! That was a bit of work, but fairly straightforward. Oh, and for `isContextual()`, `return false`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 9
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 12 - 36
    public function isContextual(Query $query): bool
    {
        return false;
    }
}
```

We won't need it, but this method is kinda cool. If you return `true`, then you can read information from the current page to *change* the query - like if you were on a "category" page and needed to list only products *in* that category.

Tagging the Query Type Handler Class

Anyways, that's *it*. This is now a functional query type handler! But if you go over and refresh... it *still* doesn't work. We get the *same* error. That's because we need to *associate* this query type handler class with the `latest_recipes` query type in our config. To do that, we need to give the service a tag... and there's a really cool way to do this thanks to Symfony 6.1.

Above the class, add an attribute called `#[AutoconfigureTag()]`. The name of the tag we need is `netgen_layouts.query_type_handler`: this is straight out of the documentation. We

also need to pass an array with a `type` key set to `latest_recipes`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
↕ // ... lines 1 - 8
9  use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;
10
11 #[AutoconfigureTag('netgen_layouts.query_type_handler', ['type' =>
   'latest_recipes'])]
12 class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
13 {
↕ // ... lines 14 - 42
43 }
```

This `type` must match what we have in our config:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
↕ // ... lines 2 - 8
9      query_types:
10         latest_recipes:
↕ // ... lines 11 - 12
```

It ties the two together.

And now... the page *works*! If we click on our Grid block... we can switch to "Dynamic collection". *Awesome*! I'll hit Apply and... everything immediately stops loading!

When you have an error in the admin section, there's a good chance it'll show up via an AJAX call. Often, Layouts will *show* you the error in a modal. But if it doesn't, no worries: just look down here on the web debug toolbar. Yup! We have a 400 error.

Let's fix that next by creating a *value converter*. Then we'll make our query *even smarter*.

Chapter 7: Value Converter

As soon as we changed our Grid type to use a Dynamic collection... it stopped loading. The error is hiding down here in this AJAX call. The best way to see it is to open that URL in a new tab. There we go:

> *"Value converter for `App\Entity\Recipe` type does not exist."*

Okay, so far, we've created a custom "value type" for `Recipe`, which was just this config, and a custom "query type" which allows us to load a list of the latest recipes by running the query inside of the associated class. *Now* we're getting this value converter error.

Creating the Value Converter Class

A value converter is really simple: it's a class that transforms the underlying object - `Recipe` - into a format that Layouts can understand. In that same `src/Layouts/` directory, let's create a `RecipeValueConverter` class... and make it implement `ValueConverterInterface`:

src/Layouts/RecipeValueConverter.php

```
// ... lines 1 - 2
namespace App\Layouts;

use Netgen\Layouts\Item\ValueConverterInterface;

class RecipeValueConverter implements ValueConverterInterface
{
// ... lines 9 - 42
}
```

You know the drill: go to "Code" -> "Generate" (or `Command`+`N` on a Mac) and hit "Implement methods" to generate the *seven* we need:

src/Layouts/RecipeValueConverter.php

```
// ... lines 1 - 6
class RecipeValueConverter implements ValueConverterInterface
{
    public function supports(object $object): bool
    {
        // TODO: Implement supports() method.
    }

    public function getValueType(object $object): string
    {
        // TODO: Implement getValueType() method.
    }

    public function getId(object $object)
    {
        // TODO: Implement getId() method.
    }

    public function getRemoteId(object $object)
    {
        // TODO: Implement getRemoteId() method.
    }

    public function getName(object $object): string
    {
        // TODO: Implement getName() method.
    }

    public function getIsVisible(object $object): bool
    {
        // TODO: Implement getIsVisible() method.
    }

    public function getObject(object $object): object
    {
        // TODO: Implement getObject() method.
    }
}
```

I know, that sounds like a lot, but these are super easy to fill in.

First, for `supports()`, Layouts will call this method *every* time it has a "value" it doesn't understand. We want to tell it that *we* know how to convert the `$object` if it's an `instanceof Recipe`:

src/Layouts/RecipeValueConverter.php

```
// ... lines 1 - 4
use App\Entity\Recipe;
// ... lines 6 - 7
class RecipeValueConverter implements ValueConverterInterface
{
    public function supports(object $object): bool
    {
        return $object instanceof Recipe;
    }
// ... lines 14 - 45
}
```

Second, for `getValueType()`, `return` the internal *key* of our value type: `doctrine_recipe`:

src/Layouts/RecipeValueConverter.php

```
// ... lines 1 - 7
class RecipeValueConverter implements ValueConverterInterface
{
// ... lines 10 - 14
    public function getValueType(object $object): string
    {
        return 'doctrine_recipe';
    }
// ... lines 19 - 45
}
```

Next is `getId()`... and we're literally going to `return` our ID with `$object->getId()`:

src/Layouts/RecipeValueConverter.php

```
// ... lines 1 - 7
class RecipeValueConverter implements ValueConverterInterface
{
// ... lines 10 - 19
    public function getId(object $object)
    {
        return $object->getId();
    }
// ... lines 24 - 45
}
```

We don't have autocomplete on this, but *we* know this object will be a `Recipe`.

For `getRemoteId()`, just `return $this->getId($object)`:

src/Layouts/RecipeValueConverter.php

```
↕ // ... lines 1 - 7
8 class RecipeValueConverter implements ValueConverterInterface
9 {
↕ // ... lines 10 - 24
25     public function getRemoteId(object $object)
26     {
27         return $this->getId($object);
28     }
↕ // ... lines 29 - 45
46 }
```

This method is only important if you plan to use the import feature in Layouts to move data, for example, between staging and production. If *were* planning to do that, you could give your objects a UUID and return that here.

Down here, for `getName()`, this will be a human-readable name shown in the admin area. This time, to help my editor, let's `assert()` that `$object instanceof Recipe`:

src/Layouts/RecipeValueConverter.php

```
↕ // ... lines 1 - 7
8 class RecipeValueConverter implements ValueConverterInterface
9 {
↕ // ... lines 10 - 29
30     public function getName(object $object): string
31     {
32         assert($object instanceof Recipe);
↕ // ... lines 33 - 34
35     }
↕ // ... lines 36 - 45
46 }
```

Two things about this. First, *we* know that this object will always be a `Recipe` because, up in `supports()`, we *said* that's that only object we support. Second, if you haven't seen the `assert()` function before, if the `$object` is *not* an `instanceof` `Recipe`, it will throw an exception. It's a really easy way to tell your editor - or other tools like PHPStan - that the object is *definitely* an instance of `Recipe`.... which means *now* we get autocompletion when we say `return $object->getName()`:

src/Layouts/RecipeValueConverter.php

```
↕ // ... lines 1 - 7
8  class RecipeValueConverter implements ValueConverterInterface
9  {
↕ // ... lines 10 - 29
30     public function getName(object $object): string
31     {
32         assert($object instanceof Recipe);
33
34         return $object->getName();
35     }
↕ // ... lines 36 - 45
46 }
```

Next is `getIsVisible()`. Just `return true`:

src/Layouts/RecipeValueConverter.php

```
↕ // ... lines 1 - 7
8  class RecipeValueConverter implements ValueConverterInterface
9  {
↕ // ... lines 10 - 36
37     public function getIsVisible(object $object): bool
38     {
39         return true;
40     }
↕ // ... lines 41 - 45
46 }
```

If your recipes could be published or unpublished, for example, then you could check *that* here to return `true` or `false`.

Finally, for `getObject()`, `return $object`:

src/Layouts/RecipeValueConverter.php

```
↕ // ... lines 1 - 7
8  class RecipeValueConverter implements ValueConverterInterface
9  {
↕ // ... lines 10 - 41
42     public function getObject(object $object): object
43     {
44         return $object;
45     }
46 }
```

I know, that seems a bit silly, but this is a handy way for you to *change* your `Recipe` after it's loaded if you needed to. But that's not something that we *need* to do.

And... done!

This time, unlike with the query type handler, autoconfiguration takes care of *everything*... so we don't need to add a manual tag up here. Watch: move over and try refreshing the AJAX endpoint first. *That* works! Now go over, refresh the layouts admin page... and *whoa*. Check it out! We see a bunch of items on our Grid! If we click that, we see the items loading below. That's awesome!

Customizing the Item Results

Notice that we *only* had to choose "dynamic collection". We... never told the system that we wanted to use the "latest recipes" query type. That's simply because we only have *one* query type... and so Layouts guessed that's what we wanted. If we added a *second* query type to the system, we would see another select drop-down here where we could choose between latest recipes and "most popular" recipes, for example.

So this is using our "latest recipes" query type to get 25 results. If we were trying to recreate this area here, we would only want 4. So let's limit the number of items to four. Cool!

Checking out the Frontend

What does this look like on the frontend? Let's find out! Hit "publish and continue editing" and.... once that saves, go over and refresh. It should show up right here but... we see absolutely nothing! Or... it *seems* that way at first.

But when we inspect element... and zoom in a bit... there's a `div` with the class `ngl-vt-grid` on it. And inside, a row and inside of *that*, a bunch of empty divs. If you ignore the `clearfix` elements, this renders 1, 2, 3, 4 divs for our *four* items! So the items *are* rendering... they're just rendering empty.

And, that makes sense. We haven't told layouts *how* recipe items should be rendered yet. More on that in a few minutes.

Query Type Form Options (Parameters)

But before we get there, I want to make our query type a *tiny* bit fancier. On the first pass, we ignored the `buildParameters()` method. Whelp, it turns out that this is a way for us to add extra *form fields* so an admin user can pass *options* to the query.

For example, let's add an optional search term. Say `$builder->add()` passing `term` - that will be the internal name for this new parameter - then `TextType`: the one from `Netgen\Layouts`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
↕ // ... lines 1 - 8
9  use Netgen\Layouts\Parameters\ParameterType\TextType;
↕ // ... lines 10 - 12
13 class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
14 {
↕ // ... lines 15 - 18
19     public function buildParameters(ParameterBuilderInterface $builder): void
20     {
21         $builder->add('term', TextType::class);
22     }
↕ // ... lines 23 - 44
45 }
```

There are a *bunch* of other field types for URLs, dates and more.

With *just* this, when we refresh the admin section... and click down on the grid, there it is! We've got a big new box! Of course, if we *type* anything inside, nothing happens... and it also has a weird label.

Translating the Field Label

Let's fix that label first. Layouts defaults to this odd string, but it's already running this through the *translator* via a domain called `nglayouts`. So, in the `translations/` directory, create a file called `nglayouts.en.yaml`, or use whatever format you want.

Paste the label and set it to "Search term":

translations/nglayouts.en.yaml

```
1 query.latest_recipes.term: 'Search term'
```

Try the admin section now. When we click... much better! If you still see the old label, try clearing your cache:

```
symfony console cache:clear
```

Sometimes Symfony doesn't notice when you add a *new* translation file.

Using the Parameter

Ok, to *use* the search term, head over to our query type handler. The `Query` object passed to `getValues()` contains any parameters we added. *And*, I already prepared the `createQueryBuilderOrderedByNewest()` method to accept an optional search term! Pass this `$query->getParameter()`, its name - `term` - then `->getValue()`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 12
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 15 - 23
    public function getValues(Query $query, int $offset = 0, ?int $limit = null): iterable
    {
        return $this->recipeRepository->createQueryBuilderOrderedByNewest($query->getParameter('term')->getValue())
// ... lines 27 - 30
    }
// ... lines 32 - 44
}
```

Copy that and repeat it down here for the `getCount()` method:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 12
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 15 - 32
    public function getCount(Query $query): int
    {
        return $this->recipeRepository->createQueryBuilderOrderedByNewest($query->getParameter('term')->getValue())
// ... lines 36 - 38
    }
// ... lines 40 - 44
}
```

Alrighty, let's take this thing for a test drive! Refresh the Layouts area, go down here and I think that worked! It shows no items... because I used a pretty silly search term. Clear it out. We get everything. Now type just a few letters... and watch as it changes below.

Next, let's teach layouts how to *render* recipe items both on the frontend as well as for the admin-area preview.

Chapter 8: Item View Template

Okay, team, things are looking good. We created a `Recipe` "value type", a custom query to load them, and a value converter to help layouts *understand* our `Recipe` objects.

What we have *not* done yet is tell Layouts how to *render* a `Recipe` *item*, item being the word Layouts uses for the individual "things" that grid and list blocks render. And actually, we need to tell Layouts both how to render an *admin* version of a recipe item, which will show up here, as well as the more-important frontend version of the item.

Adding an Item View

How an item is rendered is called an "item view". To add a *new* item view, we'll start in the config. Add a `view` key with `item_view` below it and *app* below that. I'll add a comment, because, in Layouts, `app` means "admin". So what we're about to define under the `app` key will be the *admin* view for our recipe item:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
            # app = admin
            app:
// ... lines 17 - 22
```

Next, add `recipes_app`... with a little note to say that this key is *not* important:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
            # app = admin
            app:
                # this key is not important
                recipes_app:
// ... lines 19 - 22
```

Unlike other things, such as `latest_recipes`, this internal key won't be used *anywhere*. Below, we need two important things. First, `template` - don't include the "s" like I did - set to a template path, like `nglayouts/` - that's a standard directory name to use for templates, but you could use anything - then, how about `admin/recipe_item.html.twig`:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
            # app = admin
            app:
                # this key is not important
                recipes_app:
                    template: 'nglayouts/admin/recipe_item.html.twig'
// ... lines 20 - 22
```

The second important thing is the very special `match` key. We need to tell Layouts that this is the template that should be used when a *recipe* item is being rendered. For example, imagine if we had *two* value types: recipes and also blog posts. Well, layouts would need to know that this is the template to use for *recipes*... but to use a different item template for blog posts.

The "match" Config Key

To do that, we'll use a strange syntax: `item\value_type` set to `doctrine_recipe`:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
            # app = admin
            app:
                # this key is not important
                recipes_app:
                    template: 'nglayouts/admin/recipe_item.html.twig'
                    match:
                        item\value_type: 'doctrine_recipe'
```

Where `doctrine_recipe` references the name of our value type up here:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 3
    value_types:
        doctrine_recipe:
// ... lines 6 - 22
```

We're going to see this `match` key several more times in this tutorial. Layouts has a bunch of built-in "matchers", which are identified by strings like `item\value_type`. These are used to help match one piece of config, like this template, with some *other* piece of config, like the `doctrine_recipe` value type. There are a *finite* number of these matchers, and we're going to see the most important ones along the way. So don't worry too much about them.

Oh, but let me fix my typo: this should be `template` with no "s".

The Two View Types: item_view & block_view

Anyways, I want to mention one quick thing about the `view` config key: there is only a small number of *sub-keys* that go under it.

Find your terminal and run:

```
php ./bin/console debug:config netgen_layouts view
```

This will dump a huge list of config, but don't be overwhelmed! We'll check out the important parts of this later. What I want you to look at are the root keys that go below `view`, like `block_view` and `layout_view`.

It turns out that there are *six* different keys you are allowed to put below the `view` key in your config, but we only care about *two* of them... which is why I'm mentioning this. When it comes to customizing your views, it's really quite simple! The first key we care about is `item_view`, which controls the templates used when rendering "items": so when rendering things inside of a grid or list. The only *other* sub-key we care about is `block_view`, which is how you configure the template used to render different block types, like the `title` block or the `text` block.

Yup, you're either rendering a *block* and want to customize its template or you're rendering an item *inside* of a block and you want to customize the template for that item. So the configuration looks gigantic, but most of these things are internal and you'll never need to worry about them.

Creating the Admin Template

Ok: we have our `item_view` for our `doctrine_recipe` for the *admin* area. Let's go add that template. In the `templates/` directory, create two new sub-directories: `nglayouts/admin/`. And then, a new file called `recipe_item.html.twig`. Inside, write `Does it work?` and... let's also use the `dump()` function so we can see what *variables* we have access to:

templates/nglayouts/admin/recipe_item.html.twig

```
1 Does it work?
2 {{ dump() }}
```

Alright, head back to your browser, refresh the layouts admin and... it *does* work! *And*, apparently, we have access to several variables. The most important is `item`. This is a `CmsItem` object from Layouts... and it has a property called `object` set to our `Recipe`!

Let's use that! Say `{{ item.object.name }}`, then a pipe, and... let's also print a date: `{{ item.object.createdAt }}` - one of the other properties on `Recipe` piped into the `date` filter with `Y-m-d`:

templates/nglayouts/admin/recipe_item.html.twig

```
1 {{ item.object.name }} | <time>{{ item.object.createdAt|date('Y-m-d') }}</time>
```

Let's check it! Move over, refresh and... got it! You can make this fancier if you want, but that'll work for us.

Next: let's create the frontend item view.

Chapter 9: Frontend Item View

Time to create the `Recipe` item view for the *frontend*. This starts almost exactly the same. In fact, copy the admin config... then paste. In Layouts, we know that the `app` key means the "admin" section. And, it turns out, `default` is used to mean the *frontend*:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
                # this key is not important
                recipes_default:
// ... lines 26 - 29
```

Frontend (default) item_view & Template

Once again, this internal name isn't important, for the template, use the same path but `frontend`... and keep `match` exactly the same:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
                # this key is not important
                recipes_default:
                    template: 'nglayouts/frontend/recipe_item.html.twig'
                    match:
                        item\value_type: 'doctrine_recipe'
```

I love when things are boring and easy! Let's go create that template. In `nglayouts/`, make the `frontend/` directory... and inside, `recipe_item.html.twig`.

Layouts will pass this the *same* variables as the admin item template. This means we can, once again, use `{{ item.object }}` to access our `Recipe` object. Let's print the `name` key to see if things are working:

templates/nglayouts/frontend/recipe_item.html.twig

```
1 {{ item.object.name }}
```

And... they *are* working. It's alive!

Checking Templates in the Twig Profiler

One of my favorite things to do when I start working with templates inside Layouts is to click the Twig item on the web debug toolbar. Here, we can actually *see* how Layouts is rendering. Yup, it renders `layout_2.html.twig`... then starts rendering each zone. It renders our `navigation` block, the `hero` block, then, eventually down here, the grid. You can see it's using `grid/3_columns.html.twig`. This is something we can control in the admin area. Click the grid. On the right, we're looking at the "Content" tab. But there's also a "Design" tab. Change this to "4 columns"... and I'll hit "Publish and continue editing".

If we refreshed now and reloaded the Twig profiler, we would see it rendering `4_columns.html.twig`. Then, hey! Inside of each column, it renders *our* `recipe_item.html.twig`. This is just really cool to see, and we're going to look at this again later when we talk about *overriding* core templates.

Bootstrap 4 CSS

One thing I *do* need to mention is that our app is using Bootstrap version 4, not Bootstrap 5. The reason is because, right now, the grid template renders Bootstrap version *4* markup. If you wanted to use Bootstrap 5, that's totally possible, but you would need to override these columns templates - like `4_columns.html.twig` - to tweak the classes. Overriding core templates is actually *super* easy, and we'll talk about how to do it soon.

Customizing our Frontend Template

Ok, let's bring this frontend view to life! Open up the homepage template: `main/homepage.html.twig`... and scroll up to where we loop over the latest recipes. Perfect. What I basically want to do is *steal* the markup for one of these recipe tiles... then paste that into the frontend template:

templates/nglayouts/frontend/recipe_item.html.twig

```
<a href="{{ path('app_recipes_show', { slug: recipe.slug }) }}" class="text-center recipe-container p-3">
    <div class="p-3 entity-img">
        <img src="{{ asset(recipe.imageUrl) }}" width="auto" height="115" alt="{{ recipe.name }} image">
    </div>
    <h3 class="mt-3">{{ recipe.name }}</h3>
    <small>{{ recipe.timeAsWords }} (prep & cook)</small>
</a>
```

Now we just need to tweak some variables: instead of `recipe.slug`, it needs to be `item.object.slug`. I'll do a find and replace: replace `recipe.` with `item.object.`:

templates/nglayouts/frontend/recipe_item.html.twig

```
<a href="{{ path('app_recipes_show', { slug: item.object.slug }) }}" class="text-center recipe-container p-3">
    <div class="p-3 entity-img">
        <img src="{{ asset(item.object.imageUrl) }}" width="auto" height="115" alt="{{ item.object.name }} image">
    </div>
    <h3 class="mt-3">{{ item.object.name }}</h3>
    <small>{{ item.object.timeAsWords }} (prep & cook)</small>
</a>
```

Wrapping Blocks in a Container

Nice! Let's see if that worked. Move over, refresh... and it did! That looks like the frontend. We're awesome! Except, it's missing the "gutter" that we have in the original. Inspect element. Ah, the difference is that the original columns were inside of a `container` div, which adds the margin. In the new code, we *are* inside of a row... but not a `container`.

To fix this in Layouts, let's add our favorite utility block: a column! Move the grid *into* that column. Then, we *could* add a CSS class like we did before in the hero area. But instead, take a shortcut and check "Wrap in container".

Hit "Publish and continue editing" and refresh. Whoops - wrong page. Head back to the homepage and... it looks great! It's now inside of an element with a `container` class!

This "Wrap in container" is *super* handy: it literally adds an extra `div` around your block with `class="container"` and *every* block supports this. Heck, we didn't even *need* a column: we could have just checked the "Wrap in container" on the grid itself.

The only reason I put this inside of a *column* is so we can *also* add the "Latest Recipes" header there too. Drag a new "Title" block into the column. Get outta here Apple! Inside, type "Latest Recipes" and change to an `h2`.

Hit our favorite "Publish and continue editing", refresh and... even closer! We just need to center this... and maybe give it a little top margin. Add two classes to the title: `text-center` and `my-5` for some vertical margin: both classes come from Bootstrap. I'm just repeating the classes that my designer was already using in the template.

Publish that... and when we try it... it matches *exactly*. Woo! But *now*, we have full control over the recipes inside! We could change to a different query, change the *number* of items or, in a little while, we could choose to *manually* select the *exact* recipes to show. We can also now embed lists and grids of recipes *anywhere* we want on the site.

Cleanup!

To celebrate, remove the *entire* `latest_recipes` Twig block:

templates/main/homepage.html.twig

```
// ... lines 1 - 14
{% block latest_recipes %}
    <div class="container">
        <h2 class="text-center my-5">Latest Recipes</h2>
        <div class="row">
            {% for recipe in latestRecipes %}
                <div class="col-3">
                    <a href="{{ path('app_recipes_show', { slug: recipe.slug }) }}" class="text-center recipe-container p-3">
                        <div class="p-3 entity-img">
                            <img src="{{ asset(recipe.imageUrl) }}" width="auto" height="115" alt="{{ recipe.name }} image">
                        </div>
                        <h3 class="mt-3">{{ recipe.name }}</h3>
                        <small>{{ recipe.timeAsWords }} (prep & cook)</small>
                    </a>
                </div>
            {% endfor %}
        </div>
        <div class="text-center mt-5 text-underline"><u><a href="#">Show More</a></u></div>
    </div>
{% endblock %}
// ... lines 34 - 70
```

And, up in `MainController`, delete the query, the variable, the repository argument and the `use` statement:

src/Controller/MainController.php

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

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class MainController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(): Response
    {
        return $this->render('main/homepage.html.twig', [
        ]);
    }
}
```

When we refresh, we have just *one* "Latest Recipes" section coming from our dynamic block. Oh, but notice in the layouts admin, we're *still* rendering the `latest_recipes` block... even though it doesn't exist anymore! Layouts is pretty forgiving to admin users: instead of throwing an error, it simply renders nothing.

But let's delete that... then publish... and take one last look. I love it!

Next: now that we have this grid inside of layouts, we can do some cool stuff with it, like enabling Ajax-powered pagination.

Chapter 10: Ajax Pagination & CSS/JS

Now that we're rendering these recipe items via the grid block type, check out what we can do. Click the grid, go to the design tab and then check "Enable pagination". Then you can choose between a pager style with page links, like 1, 2, 3 and 4, or just a "load more" button. Let's use that one.

All right, hit "Publish and continue editing". Then... once that saves, refresh to see... absolutely nothing! The pagination is powered entirely via JavaScript and Ajax. And we don't *see* anything because we haven't, yet, included the JavaScript needed onto our page.

Including the CSS/JS Templates

Adding it is pretty easy. Go to `templates/base.html.twig`. Up here in the `head` area, we're going to include two templates. The first is: `@NetgenLayoutsStandard/page_head.html.twig`... and pass this an extra variable: `full: true`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 7
        {{ include('@NetgenLayoutsStandard/page_head.html.twig', { full: true }) }}
// ... lines 9 - 16
    </head>
// ... lines 18 - 69
</html>
```

This will load the CSS and JavaScript that support these gallery items down here. I'm not gonna talk about these gallery blocks in this tutorial, but they're basically like a list or grid block, except that they have JavaScript to turn them into sliders or thumb galleries.

So this includes the CSS and JavaScript for those, as well as a small grid CSS file to help render the grid columns on your page in case you don't have Bootstrap. The `full: true` tells it

to bring in jQuery as well as two other JavaScript libraries called `magnific-popup` and `swiper`. All of these are needed for those gallery blocks.

So, yes, if you're not using one of those gallery blocks, you could avoid including this file entirely. But I'll leave it.

But notice, I didn't say anything about pagination. For that, we need to include a second template. Copy this line, paste, remove the word `Standard` and this doesn't need the `full` variable:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 7
        {{ include('@NetgenLayoutsStandard/page_head.html.twig', { full: true }) }}
        {{ include('@NetgenLayouts/page_head.html.twig') }}
// ... lines 10 - 16
    </head>
// ... lines 18 - 69
</html>
```

This template is dead simple: it brings in a tiny bit of CSS and a little bit of JavaScript to power Ajax pagination. And these are the *only* two templates that you'll ever need to include for Layouts JavaScript and CSS.

Adding the "ajax" Item Template

Refresh and... there it is! And when we click the new link... it explodes with a 500 error! Whoops.

Open that URL in a new tab. Interesting:

> *“No template match could be found for "item_view" view and content "ajax".”*

When we click "Load more", no surprise, that Ajax call renders the next recipe *items*. You might *think* that this would re-use our "frontend" item view template. But... there's actually a different section specifically for when content is rendered via Ajax. Copy the `default` frontend section entirely, paste, then change it to `ajax`:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 29
            ajax:
                # this key is not important
                latest_recipes_default:
                    template: 'nglayouts/frontend/recipe_item.html.twig'
                    match:
                        item\value_type: 'doctrine_recipe'
```

Nothing else needs to change: when we're in `ajax` mode, use the normal frontend template.

Now, if we refresh the Ajax endpoint... it works! Reload the homepage and click "Load more". That is *so* nice!

Translating the Pagination Button

Though, *minor* thing, our designers *really* want to use the text "Show More". No problem: *everything* that Layouts renders is processed through the translator. Click the translation icon on the web debug toolbar. Oh, there it is! Apparently the translation key is `collection.pager.load_more`.

Copy that... then go open our translation file - `nglayouts.en.yaml` - and paste. My editor changed the format... which actually *would* work... but I'll go back to the flatter format. Set this to "Show More":

translations/nglayouts.en.yaml

```
// ... line 1
collection.pager.load_more: 'Show More'
```

Spin over and... we got it!

CSS Changes to Pagination

Ok, *one* more change to make our designers happy. Inspect element on the button. Layouts adds a bunch of classes, which are styled via that CSS we included. *And*, of course, we can override that if needed.

In our editor, open `assets/styles/app.css`. As a reminder, we're already running Webpack Encore in the background. So, if we change this file, that change will automatically be rebuilt and used on the frontend.

At the bottom, I'll paste some CSS to give that button more margin but no border:

assets/styles/app.css

```
// ... lines 1 - 101
.ajax-navigation {
    margin-top: 2rem;
}
.ajax-load-more {
    border: none;
}
```

Flip back over, refresh and... our designers are happy.

So thanks to layouts, we get *free* Ajax pagination, which we can pretty easily customize. That's sweet.

Grids vs Custom Twig Content

At this point, because we're able to render grids and lists of recipes, we *could* go into the "Recipes List" layout and *replace* this hardcoded HTML, which comes from the template: `templates/recipes/list.html.twig`. Yup, we could, in theory, remove this and replace it with a list block.

The only problem... is that it wouldn't look quite right. Instead of rendering like it does now, Layouts would use our *item* template: so each item would look like it does on the homepage.

Now, we *can* fix that by creating a *second* way to render recipe items, and we *will* talk about that later. But I'm bringing this up for an important reason. Unless we're planning to reuse this list and how it looks on *other* pages on our site, there's no huge benefit to doing the work to convert it into something that we can render via Layouts. Since it's only used here, rendering it via Twig is perfectly fine.

Next: let's improve the recipe system by making it possible to *manually* select items.

Chapter 11: Content Browser

We can *now* embed lists, grids, or thumb galleries of recipes into *any* layout dynamically. That's *super* cool! And we could always create more query types to, for example, choose between the *latest* recipes or *most popular* recipes.

But what about being able to *manually* select recipes? Maybe we want to feature four *specific* recipes on the homepage. In the Layouts area, on the grid, if you change the "Collection type", we *can* switch to "Manual collection". But then... we can't actually select any items.

Enabling Manual Items in the Config

To allow items (in our case, *recipes*) to be selected manually, we first need to allow that in the config. Earlier, when we created the `value_types` config, we set `manual_items` to `false`. Change that to `true`:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 3
4      value_types:
5          doctrine_recipe:
↕  // ... line 6
7              manual_items: true
↕  // ... lines 8 - 36
```

And now, when we try the page, we're greeted with an error!

> *"Netgen Content Browser backend for `doctrine_recipe` value type does not exist."*

Yep! We need to implement a class that helps Layouts *browse* our recipes. That's called a "content browser".

Configuring the "item type" in NetgenContentBrowserBundle

Adding a content browser is actually done by a *completely* different bundle, which you can use *outside* of Netgen Layouts. It's handy if you need a nice interface for browsing and selecting items.

Since the content browser lives in a different bundle, it's not *required*, but I'm going to configure this with a new config file called `netgen_content_browser.yaml`. Inside, set the root key to `netgen_content_browser` to configure the "NetgenContentBrowserBundle":

config/packages/netgen_content_browser.yaml

```
1  netgen_content_browser:
↕  // ... lines 2 - 8
```

Inside of *this*, we get to describe all of the different "manual things" that we want to be able to browse. To do that, add an `item_types` key, and, for the first item, go grab the value type's internal name - `doctrine_recipe` - so that these match, paste, then give this a name. How about... `Recipes` with a cute strawberry icon:

config/packages/netgen_content_browser.yaml

```
1  netgen_content_browser:
2      item_types:
3          # must match "value_types" key in netgen_layouts config
4          doctrine_recipe:
5              name: 'Recipes 🍓'
↕  // ... lines 6 - 8
```

The only other thing we need here is a `preview` key with a `template` sub-key, which I'll set to `nglayouts/content_browser/recipe_preview.html.twig`:

config/packages/netgen_content_browser.yaml

```
1  netgen_content_browser:
2      item_types:
3          # must match "value_types" key in netgen_layouts config
4          doctrine_recipe:
5              name: 'Recipes 🍓'
6              preview:
7                  template: 'nglayouts/content_browser/recipe_preview.html.twig'
```

Oh! And make sure you spell "template" correctly. Whoops! Anyways, we're setting this `preview.template` because the configuration *requires* us to... but we'll worry about *creating* that template later.

Creating the Backend Class

If we head over and refresh... we get the *same* error. That's because we need a backend class that will *connect* to this new item type. Creating a backend is a simple process, but it *does* require a few different classes.

In the `src/` directory, let's create a new directory called `ContentBrowser/`... and inside of that, a PHP class called `RecipeBrowserBackend`. This needs to implement `BackendInterface`: the one from `Netgen\ContentBrowser\Backend`:

src/ContentBrowser/RecipeBrowserBackend.php

```
↕ // ... lines 1 - 2
3 namespace App\ContentBrowser;
4
5 use Netgen\ContentBrowser\Backend\BackendInterface;
↕ // ... lines 6 - 8
9 class RecipeBrowserBackend implements BackendInterface
10 {
↕ // ... lines 11 - 54
55 }
```

Then, go to "Code"->"Generate" (or `Command`+`N` on a Mac) to implement the *nine* methods this needs! Don't worry: it's not as bad as it looks:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 2
namespace App\ContentBrowser;

use Netgen\ContentBrowser\Backend\BackendInterface;
use Netgen\ContentBrowser\Item\ItemInterface;
use Netgen\ContentBrowser\Item\LocationInterface;

class RecipeBrowserBackend implements BackendInterface
{
    public function getSections(): iterable
    {
        // TODO: Implement getSections() method.
    }

    public function loadLocation($id): LocationInterface
    {
        // TODO: Implement loadLocation() method.
    }

    public function loadItem($value): ItemInterface
    {
        // TODO: Implement loadItem() method.
    }

    public function getSubLocations(LocationInterface $location): iterable
    {
        // TODO: Implement getSubLocations() method.
    }

    public function getSubLocationsCount(LocationInterface $location): int
    {
        // TODO: Implement getSubLocationsCount() method.
    }

    public function getSubItems(LocationInterface $location, int $offset = 0, int $limit = 25): iterable
    {
        // TODO: Implement getSubItems() method.
    }

    public function getSubItemsCount(LocationInterface $location): int
    {
        // TODO: Implement getSubItemsCount() method.
    }

```

```
46      public function search(string $searchText, int $offset = 0, int $limit = 25): iterable
47      {
48          // TODO: Implement search() method.
49      }
50
51      public function searchCount(string $searchText): int
52      {
53          // TODO: Implement searchCount() method.
54      }
55  }
```

Finally, to *link* this backend class to the item type in our config, we need to give this service a tag. We'll do this the same way we did earlier for the query type: with `AutoconfigureTag`. In fact, I'll steal this `AutoconfigureTag` since I'm here... paste that... and add the `use` statement for it. This time, the tag name is `netgen_content_browser.backend`, and instead of `type`, use `item_type`. Set this to the key we have in the config: `doctrine_recipe`. Paste and... cool!

src/ContentBrowser/RecipeBrowserBackend.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;
9
10 #[AutoconfigureTag('netgen_content_browser.backend', [ 'item_type' => 'doctrine_recipe' ])]
11 class RecipeBrowserBackend implements BackendInterface
12 {
↕ // ... lines 13 - 56
57 }
```

This time when we refresh... the error is *gone*. Let's temporarily add a new Grid to the layout... and choose "Manual collection". Now... check it out! Because we have a backend, we see an "Add items" button! And when we click it... it *fails*. That shouldn't be too surprising... since our backend class is still completely empty. If you want to see the *exact* error, you could open up the AJAX call.

Creating the Location Class

The content browser system works like this: in these methods, we describe a "tree structure", kind of like a filesystem. "Locations" are like directories and "items" are like the "files", or, in our case, the individual recipes.

We're going to keep things really simple. Instead of having different "directories" or "categories" of recipes that you can browse, we're going to have a single directory - or "location" - that *all* recipes will live inside. You'll see what this looks like in the UI in a few minutes.

To get this working, inside `src/ContentBrowser/`, we need to create a class that represents a location. I'll call it `BrowserRootLocation`. This class... isn't super interesting: it's just some low-level plumbing that we *must* have. Make this implement `LocationInterface`, and below, generate the three methods we need:

src/ContentBrowser/BrowserRootLocation.php

```
// ... lines 1 - 2
namespace App\ContentBrowser;

use Netgen\ContentBrowser\Item\LocationInterface;

class BrowserRootLocation implements LocationInterface
{
    public function getLocationId()
    {
        // TODO: Implement getLocationId() method.
    }

    public function getName(): string
    {
        // TODO: Implement getName() method.
    }

    public function getParentId()
    {
        // TODO: Implement getParentId() method.
    }
}
```

Again, this class will represent the one and only "location". So for `getLocationId()`, we can return *anything*. I'm going to `return 0`. You'll see how that's used in a second. For `getName()`, this is what will be displayed in the admin section. I'll `return 'All'`. And for `getParentId()`, `return null`:

src/ContentBrowser/BrowserRootLocation.php

```
// ... lines 1 - 6
class BrowserRootLocation implements LocationInterface
{
    public function getLocationId()
    {
        return 0;
    }

    public function getName(): string
    {
        return 'All';
    }

    public function getParentId()
    {
        return null;
    }
}
```

If you have a more complex system with multiple sub-directories, you could create a *hierarchy* of locations.

All right, let's update our backend class to use this. Up here, `getSections()` will be called as *soon* as the user opens up the content browser. Our job is to return all of the root "directories" - or "locations". We have just one: `return [new BrowserRootLocation()]`:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 10
class RecipeBrowserBackend implements BackendInterface
{
    public function getSections(): iterable
    {
        return [new BrowserRootLocation()];
    }
// ... lines 17 - 60
}
```

After this is called, the content browser will call `getLocationId()` on each one and make an AJAX request to get more information about them. For us, this will happen just *one* time where the ID is `0`. It looks weird, but all we need to do is return that same location: `if ($id === '0')`, then `return new BrowserRootLocation()`:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 10
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 13 - 17
    public function loadLocation($id): LocationInterface
    {
        if ($id === '0') {
            return new BrowserRootLocation();
        }
// ... lines 23 - 24
    }
// ... lines 26 - 60
}
```

Notice I'm using `'0'` as a string, but... in `getLocationId()` we returned an integer:

src/ContentBrowser/BrowserRootLocation.php

```
// ... lines 1 - 6
class BrowserRootLocation implements LocationInterface
{
    public function getLocationId()
    {
        return 0;
    }
// ... lines 13 - 22
}
```

That's because the id will be passed into JavaScript and used in an Ajax call. By the time it gets here, it'll be a string. A small detail to keep in mind.

At the end, just in case `throw` a `new \InvalidArgumentException()` and pass a message about an invalid location:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 10
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 13 - 17
    public function loadLocation($id): LocationInterface
    {
        if ($id === '0') {
            return new BrowserRootLocation();
        }

        throw new \InvalidArgumentException(sprintf('Invalid location "%s"', $id));
    }
// ... lines 26 - 60
}
```

Ok! So our backend has *one* location. For the other methods, let's return the *simplest* thing possible. Leave `loadItem()` empty for a moment, for `getSubLocations()`, `return []`, for `getSubLocationsCount()`, `return 0`, for `getSubItems()`, `return []`, for `getSubItemsCount()`, `return 0`, for `search()`, `return []`... and *finally*, for `searchCount()`, `return 0`:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 10
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 13 - 26
    public function loadItem($value): ItemInterface
    {
        // TODO: Implement loadItem() method.
    }

    public function getSubLocations(LocationInterface $location): iterable
    {
        return [];
    }

    public function getSubLocationsCount(LocationInterface $location): int
    {
        return 0;
    }

    public function getSubItems(LocationInterface $location, int $offset = 0, int $limit = 25): iterable
    {
        return [];
    }

    public function getSubItemsCount(LocationInterface $location): int
    {
        return 0;
    }

    public function search(string $searchText, int $offset = 0, int $limit = 25): iterable
    {
        return [];
    }

    public function searchCount(string $searchText): int
    {
        return 0;
    }
}
```

Phew... We'll talk about each of those methods later. *But* our backend class is at least *somewhat* functional now.

If we refresh the admin area again... click on our grid, and go to "Add Items"... *it loads*! Say "hello" to the content browser! It's currently *empty*, but you can see the "All", which is from our *one* location. There are no items inside yet... because we need to return them from `getSubItems()`. Let's do that *next*

Chapter 12: Content Browser: Returning the Items

Our Content Browser is *sort of* working. We can see our one location... we just don't have any results yet. That's because, for whatever location is selected, the Content Browser calls `getSubItems()`. Our job here is to return the results. In this case, all of our recipes. If we had *multiple* locations, like recipes divided into categories, we could use the `$location` variable to return the subset. But we'll query and return *all* recipes.

Querying in getSubItems()

To do that, go to the top of the class and create a constructor with `private RecipeRepository $recipeRepository`:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 4
use App\Repository\RecipeRepository;
// ... lines 6 - 11
class RecipeBrowserBackend implements BackendInterface
{
    public function __construct(private RecipeRepository $recipeRepository)
    {
    }
// ... lines 17 - 70
}
```

Then, down here in `getSubItems()`, say `$recipes = $this->recipeRepository` and use that same method from earlier: `->createQueryBuilderOrderedByNewest()`. Below add `->setFirstResult($offset)`... and `->setMaxResults($limit)`. The Content Browser comes with pagination built-in. It passes us the offset and limit for whatever page the user is on, we plug it into the query, and everyone is happy. Finish with `getQuery()` and `getResult()`:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 11
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 14 - 46
    public function getSubItems(LocationInterface $location, int $offset = 0, int $limit = 25): iterable
    {
        $recipes = $this->recipeRepository
            ->createQueryBuilderOrderedByNewest()
            ->setFirstResult($offset)
            ->setMaxResults($limit)
            ->getQuery()
            ->getResult();
    }
// ... lines 56 - 70
}
```

Notice that `getSubItems()` returns an `iterable`... *actually* it's supposed to be an iterable of something called an `ItemInterface`. So we can't *just* return these `Recipe` objects.

Creating the ItemInterface Wrapper Class

Instead, in `src/ContentBrowser/`, create another class called, how about `RecipeBrowserItem`. Make this implement `ItemInterface` - the one from `Netgen\ContentBrowser` - then generate the four methods it needs:

src/ContentBrowser/RecipeBrowserItem.php

```
// ... lines 1 - 2
namespace App\ContentBrowser;

use Netgen\ContentBrowser\Item\ItemInterface;

class RecipeBrowserItem implements ItemInterface
{
    public function getValue()
    {
        // TODO: Implement getValue() method.
    }

    public function getName(): string
    {
        // TODO: Implement getName() method.
    }

    public function isVisible(): bool
    {
        // TODO: Implement isVisible() method.
    }

    public function isSelectable(): bool
    {
        // TODO: Implement isSelectable() method.
    }
}
```

This class will be a tiny wrapper *around* a `Recipe` object. Watch: add a `__construct()` method with `private Recipe $recipe`:

src/ContentBrowser/RecipeBrowserItem.php

```
// ... lines 1 - 4
use App\Entity\Recipe;
// ... lines 6 - 7
class RecipeBrowserItem implements ItemInterface
{
    public function __construct(private Recipe $recipe)
    {
    }
// ... lines 13 - 32
}
```

Now, for `getValue()`, this should return the "identifier", so `return $this->recipe->getId()`. For `getName()`, we just need something visual we can

show, like `$this->recipe->getName()`. And for `isVisible()`, `return true`. That's useful if a `Recipe` could be published or unpublished. We have a similar situation with `isSelectable()`:

src/ContentBrowser/RecipeBrowserItem.php

```
// ... lines 1 - 7
class RecipeBrowserItem implements ItemInterface
{
// ... lines 10 - 13
    public function getValue()
    {
        return $this->recipe->getId();
    }

    public function getName(): string
    {
        return $this->recipe->getName();
    }

    public function isVisible(): bool
    {
        return true;
    }

    public function isSelectable(): bool
    {
        return true;
    }
}
```

If you had a set of rules where you wanted to *show* certain recipes but make them not *selectable*, you could `return false` here.

And... we're done! That was easy!

Back over in our backend class, we need to turn these `Recipe` objects into `RecipeBrowserItem` objects. We can do that with `array_map()`. I'll use the fancy `fn()` syntax again, which will receive a `Recipe $recipe` argument, followed by `=> new RecipeBrowserItem($recipe)`. For the second arg, pass `$recipes`:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 12
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 15 - 47
    public function getSubItems(LocationInterface $location, int $offset = 0, int $limit = 25): iterable
    {
// ... lines 50 - 55

        return array_map(fn(Recipe $recipe) => new RecipeBrowserItem($recipe), $recipes);
    }
// ... lines 59 - 73
}
```

This is a fancy way of saying:

> *"Loop over all the recipes in the system, create a new `RecipeBrowserItem` for each one, and return that new array of items."*

All right, let's see what this looks like! Refresh the layout, click on the Grid, go back to "Add items" and... got it! We see *ten* items!

Implementing getSubItemsCount()

But we *should* have multiple pages. Ah, that's because we're still returning `0` from `getSubItemsCount()`. Let's fix that. Steal the query from above... paste, return this, remove `setFirstResult()` and `setMaxResults()`, add `->select('COUNT(recipe.id)')`, and then call `getSingleScalarResult()` at the bottom:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 12
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 15 - 59
    public function getSubItemsCount(LocationInterface $location): int
    {
        return $this->recipeRepository
            ->createQueryBuilderOrderedByNewest()
            ->select('COUNT(recipe.id)')
            ->getQuery()
            ->getSingleScalarResult();
    }
// ... lines 68 - 77
}
```

And just like that, when we refresh... and open the Content Browser... *we have pages*!

Adding the Search Functionality

Tip

Though this solution works fine, `search()` and `searchCount()` are deprecated in favor of `searchItems()` and `searchItemsCount()`. To use the new methods, keep the old methods (because they're still part of the interface) and use the following for the new methods:

```
class RecipeBrowserBackend implements BackendInterface
{
    // ...

    public function search(string $searchText, int $offset = 0, int $limit = 25): i
    {
        // deprecated
        return [];
    }

    public function searchCount(string $searchText): int
    {
        // deprecated
        return 0;
    }

    public function searchItems(SearchQuery $searchQuery)
    {
        $recipes = $this->recipeRepository
            ->createQueryBuilderOrderedByNewest($searchQuery->getSearchText())
            ->setFirstResult($searchQuery->getOffset())
            ->setMaxResults($searchQuery->getLimit())
            ->getQuery()
            ->getResult();
        return new RecipeBrowserSearchResults($recipes);
    }

    public function searchItemsCount(SearchQuery $searchQuery)
    {
        return $this->recipeRepository
            ->createQueryBuilderOrderedByNewest($searchQuery->getSearchText())
            ->select('COUNT(recipe.id)')
            ->getQuery()
            ->getSingleScalarResult();
```

```
    }
}
```

You'll also need a new `RecipeBrowserSearchResults` class:

```
// src/ContentBrowser/RecipeBrowserSearchResults.php
namespace App\ContentBrowser;

use Netgen\ContentBrowser\Backend\SearchResultInterface;
use App\Entity\Recipe;

class RecipeBrowserSearchResults implements SearchResultInterface
{
    public function __construct(private array $results)
    {
    }

    public function getResults(): iterable
    {
        return array_map(fn (Recipe $recipe) => new RecipeBrowserItem($recipe), $th
    }
}
```

Thanks to Joris in the comments for noticing this!

Ok, but could we search for recipes? *Absolutely*. We can leverage `search()` and `searchCount()`. This is simple. Steal all of the logic from `getSubItems()`, paste into `search()` and pass `$searchText` to the QueryBuilder method, which already allows this argument:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 12
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 15 - 68
    public function search(string $searchText, int $offset = 0, int $limit = 25): iterable
    {
        $recipes = $this->recipeRepository
            ->createQueryBuilderOrderedByNewest($searchText)
            ->setFirstResult($offset)
            ->setMaxResults($limit)
            ->getQuery()
            ->getResult();

        return array_map(fn(Recipe $recipe) => new RecipeBrowserItem($recipe), $recipes);
    }
// ... lines 80 - 88
}
```

If you want to have a bit less code duplication, you could isolate this into a `private` method at the bottom.

Also copy the logic from the other count method... paste that into `searchCount()`, and pass it `$searchText` as well:

src/ContentBrowser/RecipeBrowserBackend.php

```
// ... lines 1 - 12
class RecipeBrowserBackend implements BackendInterface
{
// ... lines 15 - 80
    public function searchCount(string $searchText): int
    {
        return $this->recipeRepository
            ->createQueryBuilderOrderedByNewest($searchText)
            ->select('COUNT(recipe.id)')
            ->getQuery()
            ->getSingleScalarResult();
    }
}
```

And just like that, if we move over here and try to search... *it works*. That's awesome!

Alright - select a few items, hit "Confirm" and... oh no! It breaks! It still says "Loading". If you look down on the web debug toolbar, we have a 400 error. *Dang*. When we open that up, we see:

> *“Value loader for `doctrine_recipe` value type does not exist.”*

There's just *one* final piece we need: A very simple class called the "value loader". That's *next*.

Chapter 13: Value Loader + Preview Template

So our content browser was working beautifully... until we selected an item. At *that* time, it chose to do an odd thing: explode! The Ajax call that failed says:

> *"Value loader for `doctrine_recipe` value type does not exist."*

To review: we have a custom value type called `doctrine_recipe`, which we created so that we could add grids and lists of `Recipe` entities. For this to work, we have (1): a value converter to convert `Recipe` objects into a format understood by layouts. (2) a query type to allow us to use dynamic collections. (3) a browser backend class to allow us to select manual items. And now (4), we need a value loader that is able to take the "id" of these manually-selected items and turn them into `Recipe` objects. This will be the last "thing" we need for our value type, I promise!

Creating & Tagging the Value Loader

Inside the `src/Layouts/` directory, create a new class called `RecipeValueLoader`, make it implement `ValueLoaderInterface` and generate the two methods it needs:

src/Layouts/RecipeValueLoader.php

```
// ... lines 1 - 2
namespace App\Layouts;

use Netgen\Layouts\Item\ValueLoaderInterface;

class RecipeValueLoader implements ValueLoaderInterface
{
    public function load($id): ?object
    {
        // TODO: Implement load() method.
    }

    public function loadByRemoteId($remoteId): ?object
    {
        // TODO: Implement loadByRemoteId() method.
    }
}
```

These are pretty simple. But, before we fill them in, go back to the Ajax endpoint, and refresh to see... the exact same error. Why? Like we've seen with other things, we need to "associate" this `RecipeValueLoader` with our `doctrine_recipe` value type. How? No surprise! With a tag. Say `#[AutoconfigureTag()]` and this time it's called `netgen_layouts.cms_value_loader`. For the second argument, pass `value_type` set to `doctrine_recipe`:

src/Layouts/RecipeValueLoader.php

```
// ... lines 1 - 5
use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;

#[AutoconfigureTag('netgen_layouts.cms_value_loader', ['value_type' =>
'doctrine_recipe'])]
class RecipeValueLoader implements ValueLoaderInterface
{
// ... lines 11 - 19
}
```

Perfecto! If we reload now... better! That error is because we haven't actually filled in the logic yet.

Adding the Logic

Very simply, we need to take the ID and return the `Recipe` object. To do that, create a constructor that accepts a `RecipeRepository $recipeRepository` argument. And... let me clean things up:

src/Layouts/RecipeValueLoader.php

```
// ... lines 1 - 4
use App\Repository\RecipeRepository;
// ... lines 6 - 9
class RecipeValueLoader implements ValueLoaderInterface
{
    public function __construct(private RecipeRepository $recipeRepository)
    {
    }
// ... lines 15 - 24
}
```

Now, down here, return `$this->recipeRepository->find()` and pass `$id`. For `loadByRemoteId()`, which we only need if we're using the import feature to move content across databases, just `return $this->load($id)`:

src/Layouts/RecipeValueLoader.php

```
// ... lines 1 - 9
class RecipeValueLoader implements ValueLoaderInterface
{
// ... lines 12 - 15
    public function load($id): ?object
    {
        return $this->recipeRepository->find($id);
    }

    public function loadByRemoteId($remoteId): ?object
    {
        return $this->load($remoteId);
    }
}
```

And now... the Ajax call works! More importantly, if we refresh the entire layouts admin... yes! Look at our grid! We have four manual items! That is awesome! We can reorder these if we want, add more, remove them, whatever.

Try publishing this page and then reloading the homepage. There they are! Though our "latest recipes" are missing. Whoops! I think I accidentally changed this to a manual collection also. Change that back to a dynamic collection, looks good, publish and.... now... cool: everything is back.

Adding the Preview

So we now have the power to select manual items via the content browser... though when we originally add the config for all of this, we set a preview template... but never created it!

Let's open the content browser again. So on the manual grid, hit "Add items". The preview template powers the preview mode up here. If we click an item, it shows us a preview. Well, it *would*... except that we haven't actually *added* that template.

To get this working, we need to do two small things. First, open `RecipeBrowserBackend`. We skipped a few methods in here. For example, we skipped `getSubLocations()` and `getSubLocationsCount()` because those are only need if you have a *hierarchy* of locations.

We also skipped `loadItem()`. *This* is used for the preview. It will pass us the ID of the thing that's loaded and we need to return an `ItemInterface`. So very simply, we can return a

`new RecipeBrowserItem()` - that's the little class that wraps around the `Recipe` - passing `$this->recipeRepository->find($value)`:

src/ContentBrowser/RecipeBrowserBackend.php

```
↕ // ... lines 1 - 12
13 class RecipeBrowserBackend implements BackendInterface
14 {
↕ // ... lines 15 - 32
33     public function loadItem($value): ItemInterface
34     {
35         return new RecipeBrowserItem($this->recipeRepository->find($value));
36     }
↕ // ... lines 37 - 88
89 }
```

Cool! The only other thing we need to do is... actually create the preview template! In `templates/nglayouts/`, add a new directory called `content_browser/`, and inside, a new file called `recipe_preview.html.twig`. To start, just print the `dump()` function:

templates/nglayouts/content_browser/recipe_preview.html.twig

```
1 {{ dump() }}
```

The cool thing is, we don't even need to refresh. As long as we click on an item that we haven't *already* clicked on... it works! And look at this `item` variable: it's an instance of `RecipeBrowserItem`... so an instance of this class right here.

That's great... except that `RecipeBrowserItem` doesn't have a way for us to *get* the actual `Recipe` object. Fortunately, we can fix that ourselves. After all, this is *our* class! I'll go to "Code"->"Generate" to generate a `getRecipe()` method:

src/ContentBrowser/RecipeBrowserItem.php

```
↕ // ... lines 1 - 7
8 class RecipeBrowserItem implements ItemInterface
9 {
↕ // ... lines 10 - 33
34     public function getRecipe(): Recipe
35     {
36         return $this->recipe;
37     }
38 }
```

Now, in the template, we can say `{{ item.recipe.name }}`. And to make this fancier, add an `<img` whose `src` is set to `item.recipe.imageUrl`... also with an `alt` attribute:

templates/nglayouts/content_browser/recipe_preview.html.twig

```
<strong>{{ item.recipe.name }}</strong>
<br><br>
<img src="{{ asset(item.recipe.imageUrl) }}" alt="Recipe Image">
```

Once again, we don't need to refresh. If you click on an item that you've already previewed, it'll load it from memory. But if you click a new one... yeah! There's our preview! Pretty cool.

Ok, we are *done* with manual items, the content browser and all of this. By the way, there *is* a way to add more *columns* to this table, like filename, file size, created date, etc. We're not going to talk about that, but it's totally possible.

Status check: at this point, we have the ability to add a layout to any page, reorder the content on the page, add title, text, HTML blocks, or even lists and grids of dynamic recipes. That is a *lot* of power. Now I want *more* power! I want to make it possible to use the grid and list blocks to add *other* items to our page... items that do *not* live in our database at all. That's next.

Chapter 14: Contentful: Loading Data from an External CMS

If we added five more entities and we wanted to be able to select those as items in the Layouts admin, we could add five more value types, query types, and item views. Now that we know what we're doing, it's a pretty quick process and would give us a *lot* of power on our site.

But one of the beautiful things about Layouts is that our value types can come from *anywhere*: a Doctrine Entity, data on an external API, data in a Sylius store or from Ibexa CMS. In fact, systems like Sylius and Ibexa *already* have packages that do all of the work of integrating and adding the value types *for* you.

One of the biggest missing pieces on *our* site is the skills. The skills on the homepage are hard-coded and the "All Skills" link doesn't even go anywhere! We *could* have chosen to store these skills locally via another Doctrine Entity. But *instead*, we're going to load them from an external API via a service called "Contentful".

Hello Contentful!

I'll head over to Contentful.com and log in. This takes me to a "Contentful" space called "Bark & Bake" that I've already created. Contentful is *awesome*! It's basically a CMS as a service. It allows us to create different *types* of content called "content models". Right now, I have a content model called "Skill" and another one called "Advertisement". If we clicked on these, we could *enter* content via a super-friendly interface. I've already created 5 skills, each with a bunch of data.

So, you create and maintain your content *here*. *Then* Contentful has a restful API that we can use to *fetch* all of this.

Contentful is *cool*. But the point of this isn't to teach you about Contentful. Nope! It's to show you how we could grab content for Layouts from *anywhere*. For example, if we want to load "skills" from Contentful, we could manually create a new value type and do all the work that we did before, except making API requests to Contentful instead of querying the database.

But! We don't even need to do that! Why? Because Layouts *already* has a bundle that supports Contentful. That bundle add the value type, some query types, the item views and even the content browser integration *for* us. Woh.

Let's grab it!

Installing the Contentful Bundle

Spin over to your terminal and run:

```
composer require netgen/layouts-contentful -W
```

The `-W` is there just because, at least when recording this, Composer needs to be able to downgrade one small package to make all the dependencies happy. That flag allows it to do that.

Ok! The recipe for this package added a new config file: `config/packages/contentful.yaml`:

config/packages/contentful.yaml

```
# For the complete configuration, please visit
# https://www.contentful.com/developers/docs/php/tutorials/getting-started-with-contentful-and-symfony/
contentful:
    delivery:
        main:
            token: "%env(CONTENTFUL_ACCESS_TOKEN)%"
            space: "%env(CONTENTFUL_SPACE_ID)%"
```

And *this* reads two new environment variables... which live in `.env`:

.env

```
// ... lines 1 - 30
###> contentful/contentful-bundle ###
CONTENTFUL_SPACE_ID=cfexampleapi
CONTENTFUL_ACCESS_TOKEN=b4c0n73n7fu1
###< contentful/contentful-bundle ###
```

While we're here, let's update these values to point at *my* Contentful space. Copy the keys from the code block on this page and paste them here. Here's my `CONTENTFUL_SPACE_ID`... and my

`CONTENTFUL_ACCESS_TOKEN`, which will give us read access to my space:

.env

```
// ... lines 1 - 30
###> contentful/contentful-bundle ###
CONTENTFUL_SPACE_ID=uvx9svgj8l12
CONTENTFUL_ACCESS_TOKEN=3qgirZC8zMKQEnGgXNtrjRibdXYuhiFEBY9tHPyfjnw
###< contentful/contentful-bundle ###
```

Contentful + Layouts

Okay, the Layouts + Contentful integration give us two *very* separate things, and it's *super* important to understand the difference to keep everything clear.

First, the package adds an integration between Layouts and Contentful. This means it adds new value types, new query types, and all the other stuff we just added for Doctrine. In other words, we can *instantly* add the skills or advertisements from Contentful into list or grid blocks. That is *great*, and we'll see it soon.

The *second* thing the Contentful integration adds is *completely unrelated* to Layouts. It's dynamic routes. It adds a system so that every "item" in Contentful is available via its own URL. Literally, *all* of these skills will instantly have their own page on our site. This has *nothing* to do with Layouts, which is all about controlling the layout for *existing* pages on your site, not adding new pages.

Setting up the Dynamic Routing

But, since Contentful is a CMS, it *is* nice to have a page for each piece of content. To get the dynamic routes working, go into the `config/packages/` directory and add a new file called `cmf_routing.yaml`. CMF Routing is a package that Contentful uses behind the scenes to add the dynamic routes. I'll paste some config here:

config/packages/cmf_routing.yaml

```
cmf_routing:
    chain:
        routers_by_id:
            router.default: 200
            cmf_routing.dynamic_router: 100
    dynamic:
        default_controller: netgen_layouts.contentful.controller.view
        persistence:
            orm:
                enabled: true
```

It's ugly... but this part doesn't have anything to do with Layouts, so it doesn't matter too much. This is *all about* allowing Contentful to automatically add dynamic URLs to our site.

This routing system stores routes in the database... and that means we need some new database. Head over to your console and run:

```
symfony console make:migration
```

And... I get an error. *Rude*. Let's try clearing our cache... maybe something weird happened... or it didn't see my new config file yet.

```
php bin/console cache:clear
```

Once the cache clears... I'll make the migration again:

```
symfony console make:migration
```

This time... perfect! Open the `migrations/` directory, find that file and... it looks good!

migrations/Version20221024142326.php

```
// ... lines 1 - 12
final class Version20221024142326 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE TABLE contentful_entry (id VARCHAR(255) NOT NULL, name VARCHAR(255) NOT NULL, json LONGTEXT NOT NULL, is_published TINYINT(1) NOT NULL, is_deleted TINYINT(1) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');
        $this->addSql('CREATE TABLE contentful_entry_route (contentful_entry_id VARCHAR(255) NOT NULL, route_id INT NOT NULL, INDEX IDX_58B6BC6E877C153C (contentful_entry_id), INDEX IDX_58B6BC6E34ECB4E6 (route_id), PRIMARY KEY(contentful_entry_id, route_id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');
        $this->addSql('CREATE TABLE orm_redirects (id INT AUTO_INCREMENT NOT NULL, host VARCHAR(255) NOT NULL, schemes LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', methods LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', defaults LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', requirements LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', options LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', condition_expr VARCHAR(255) DEFAULT NULL, variable_pattern VARCHAR(255) DEFAULT NULL, staticPrefix VARCHAR(255) DEFAULT NULL, routeName VARCHAR(255) NOT NULL, uri VARCHAR(255) DEFAULT NULL, permanent TINYINT(1) NOT NULL, routeTargetId INT DEFAULT NULL, UNIQUE INDEX UNIQ_6CA17E0391F30BA8 (routeName), INDEX IDX_6CA17E034C0848C6 (routeTargetId), INDEX IDX_6CA17E03A5B5867E (staticPrefix), PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');
        $this->addSql('CREATE TABLE orm_routes (id INT AUTO_INCREMENT NOT NULL, host VARCHAR(255) NOT NULL, schemes LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', methods LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', defaults LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', requirements LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', options LONGTEXT NOT NULL COMMENT \'(DC2Type:array)\', condition_expr VARCHAR(255) DEFAULT NULL, variable_pattern VARCHAR(255) DEFAULT NULL, staticPrefix VARCHAR(255) DEFAULT NULL, name VARCHAR(255) NOT NULL, position INT NOT NULL, INDEX IDX_5793FCA5B5867E (staticPrefix), UNIQUE INDEX name_idx (name), PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');
        $this->addSql('ALTER TABLE contentful_entry_route ADD CONSTRAINT FK_58B6BC6E877C153C FOREIGN KEY (contentful_entry_id) REFERENCES contentful_entry (id) ON DELETE CASCADE');
        $this->addSql('ALTER TABLE contentful_entry_route ADD CONSTRAINT FK_58B6BC6E34ECB4E6 FOREIGN KEY (route_id) REFERENCES orm_routes (id) ON DELETE CASCADE');
```

```
29          $this->addSql('ALTER TABLE orm_redirects ADD CONSTRAINT
    FK_6CA17E034C0848C6 FOREIGN KEY (routeTargetId) REFERENCES orm_routes (id)');
30      }
↕   // ... lines 31 - 42
43  }
```

We have a few tables that hold info about our Contentful data... and a few to store those dynamic routes.

Now run:

```
symfony console doctrine:migrations:migrate
```

And... woohoo! We have the new tables we need.

Finally, we can run a command to *load* all of our content from Contentful and create those dynamic routes. Once again, this is functionality that has *nothing* to do with Layouts. Run:

```
symfony console contentful:sync
```

And... beautiful! It loaded six items. On production you can set up a webhook so your site is *instantly* synced with any changes that you make on Contentful. But while we're *developing*, running this command works fine.

The result of this command is that every piece of content on Contentful now has its own page! To see them, run:

```
symfony console contentful:routes
```

And... *awesome*! Apparently I have a URL called `/mashing`. Let's go check it out! Go back to our site, navigate to `/mashing` and... it works! *Sort of*. It's *here*, but the middle of it is empty.

Let's talk about what's going on next and how we can leverage Layouts to bring this page to life.

Chapter 15: Mapping a Layout to Contentful Pages

The Contentful integration we just installed added two things to our site. First, it added a Layouts integration: new value types, query types, etc so that we can select our Contentful content in list and grid blocks. Second, it added the ability for *every* piece of content on Contentful to have its own URL and page on our site. The second part has *nothing* to do with Layouts.

A minute ago, we used this handy dandy `contentful:routes` command to see that there should now be a page at the URL `/mashing`. When we went there, it didn't give us a 404 error, but it *didn't* exactly work. The page is nearly empty.

Debugging How the Dynamic Contentful Pages Work

Let's see what's going on. Click the Twig icon in the web debug toolbar to find out what templates are being rendered. Let's see here... if we go down a bit... it apparently renders `@NetgenLayoutsContentful/contentful/content.html.twig`. That must be the template for this page! Let's go check it out.

I'll hit `Shift`+`Shift` and search for `content.html.twig`: we want the one from `layouts-contentful`. And... *cool*! This is the template that's rendering that page. It prints `content.name`... but we never actually *see* that. Ah, that's because it renders it into a `block` called `content`. This eventually extends `base.html.twig`... and since our base template never *renders* `block content`, we see *nothing*. Again, this part of Contentful where you get a URL that renders a controller, that *then* renders this template... has *nothing* to do with Layouts. It's just a nice way to expose every piece of Contentful content as a page on our site.

So, *unrelated* to Layouts, if we wanted to, we could override this template in our app and customize it to work. We could change it to use `block body` and leverage this `content` variable, which represents the piece of content, to render all of the different fields.

But... *hold on* a second. Isn't that the whole point of Layouts? Layouts lets us build pages dynamically, instead of writing them entirely in Twig. Right now, this page is *not* linked to a

layout. But if we *did* link it, we could start building the page using data from the matching Contentful Skill, in this case, from the "Mashing" Skill.

Mapping a Layout to the Dynamic Page

Head over to our admin section, publish that layout, and then create a *new* layout. I'll call it "Individual Skill Layout"... and choose "Layout 2". Eventually, we'll make this look more like "Layout 5"... but we can do that later via the column blocks. That's one of the reasons why I really like "Layout 2": it's simple enough, and we can make it more complex *later* with the tools we already have.

Okay, start like normal. Close the web debug toolbar so we can link the header to the shared header... and our footer to the shared footer. *Awesome*. Then, just to get rolling, add a Title block, type something... then publish the layout.

Mapping a Layout to Contentful Entries

Next, we need to map this layout to that page. So far, we've mapped layouts either by the route name or the URL, also known as the "Path info". We *could* do that again here. But, as you'll see, what we *really* want to do is use this layout for *all* Skills pages. In a few minutes, we're going change the URL for these page from something like `/mashing` to `/skills/mashing`. When we do that (let me add a new mapping here and hit details), we *could* then use the "Path info prefix" to map this layout to any URL that starts with `/skills/`.

But, one thing that can be added to Layouts is another way to map or resolve which layout should be used on which page. And, yea! The Contentful bundle added two new ones: Contentful Entry and Contentful Space. When we go to one of these Contentful pages, the dynamic route tells Symfony which Contentful Entry - that's the individual piece of content on Contentful - *and* which Contentful Space that this page maps to.

Thanks to this, we can leverage one of these new targets to match the entry or space. For example, we could use Contentful Entry to map a specific layout to a specific *item* on Contentful. Literally, we could say:

> *“If the current Content is specifically this "Mashing" skill, then use this layout.”*

Or, we could do what I'm going to do: map via the Contentful Space. We only have one Space, so it's pretty easy. Basically, we're saying:

> *“If we are on any dynamic Contentful page, I want you to map to this layout.”*

Let's save this... then link this layout to the "Individual Skill Layout". Hit "Confirm" and... good! Head over, refresh and... it works! Yes!

Mapping to a Specific Content Type

As I mentioned earlier, we actually have *two* types of content in Contentful: *Skills* and *Advertisements*. Advertisements aren't meant to have their own page - only skills are. We're going to embed advertisements onto some *existing* pages a bit later.

Head back to the layout linking details. In addition to the Contentful Space, we can go down here to a list of conditions and select "Contentful content types". Conditions are a way to make your matching more specific. Add that condition. And, this is a bit hard to see, but we can select "Skill" or "Advertisement". Select "Skill", save changes, and... sweet! Now this will *only* match if we go to a Contentful URL that is rendering a *skill*.

At the command line, you can see that we *do* have one advertisement... it's this funny-looking URL. Yes, right now, the advertisement *is* available as a page on our site. We'll fix that in a few minutes. But, at the very least, if we went to that weird URL, the page *would* work... but wouldn't match any layout thanks to our mapping. So, it would basically be blank.

So we *now* have control over Contentful pages. Cool! Though... all we're rendering is a manual title. Snooze.

Next: Let's make our layout smarter by rendering *real* content from the matching skill.

Chapter 16: Building the Contentful Page

We now have full control over how the Contentful pages render. That's thanks to the "Individual Skill layout" that we mapped to all Contentful "Skill" pages.

But... all we have is this manual `h1` title. How can we render the real *data* for whichever Contentful Skill we're viewing?

First, on Contentful's site, if I navigate to "Content model" and click on "Skill", you can see that every Skill has 5 fields... and each field has an internal name. It's... almost easier to see this via the JSON preview. Here we go. So there's a "Title" field, it's internal name is `title`, "Short Description", "Technique", and a few other like "Image" and "Advertisement". Advertisement is actually a link to that *other* type of content.

Using the "Contentful Entry Field" Block Type

Anyways, what we *really* want to do up here is print the *skill's* title in the `h1`. Fortunately, that *is* possible, thanks to a new *block* type that the Contentful bundle added. It's here at the bottom: "Contentful entry field".

This allows us to render a single field from whatever Contentful entry is currently being rendered. Let's try it! Then delete the old `h1`.

The new block has one super important option: field identifier. Set that to the internal name of the field: `title`. And make this an `h1`. As usual, the block label is optional, but I'll include it.

Cool! Hit publish and continue editing, move over and... yes! It's dynamic. If we go to the URL for some *other* skill, like `/basic-chop`, that works too!

Adding the Fancy Hero Area

So let's get fancier. Add a column... and move this title inside. Can you guess what I'm about to do? Give the column that same `hero-wrapper` class that we used earlier. And you know what else? Each skill has a "Short description". Lets add another entry field block right below.

Notice that one option for this block is "view type". We're going to talk more about that soon, but this should match the "type" of the content that you're pulling from Contentful. So far, both `title` and this `shortDescription` are "string" types. Leave this as `div`.

Testing timer! Hit "Publish and continue editing". And... let's see how it looks. I love that! Let's add more!

Adding a Contentful Image

Every skill has an image. Inside of that same hero column, add another Contentful entry block at the bottom. This will be called `image`... and the *type* is "referenced assets". You *do* need to set a width and height. Let's do 200 by 200. Publish that... refresh and... we're on a roll!

One last thing: rendering the skill content *below* everything. By the way, we could render this in the same zone... or use the zone below. Zones don't matter much in most case.

Using a 2-Column Block

But let's make this spot more interesting. I want to render the skill content on the left and an advertisement on the right. To do that, for the first time, use a 2-column block. Set this to 66, 33 so that the left side takes up most of the space. Add a title to the left side and make it an `h3` with the text "The Technique:". Below, drag over a contentful entry field.

This one... if I go check my fields... is called `technique` and it holds rich text. If you modified it in Contentful, you'd see a rich text editor... and the final value is HTML. So, type `technique`, keep it as a `div` and select `Richtext`.

Rendering a Related Contentful Entry

Finally, on the right side, add one more Contentful entry field. Look back at the content model for Skills... and scroll down a bit. The one we want to use is called `advertisement`, and this is a "Referenced entry" type. Yup, if you edited a skill, you would choose the Advertisement from the list of Advertisements we have in Contentful. It's like a database relation.

Anyways, enter `advertisement`, hit "Publish and continue editing"... refresh and... ok! *Sort* of awesome. We need a container to bring those in. We already have a column, so click "Wrap in

container".

And... yea... though this could also use some top margin. On that same column, add a class: `my-3`. Publish this... and reload. *So* much better! Though, the Advertisement is just printing a URL... not rendering an ad. That's because Contentful doesn't *know* how to render the "Advertisement" content type. We'll help it soon.

But first, let's fix our Skill pages by prefixing all URLs with `/skills`.

Chapter 17: Customizing the Contentful Slugger

Before we go further into customizing the look and feel of our site, I want to fix the skill URLs so that instead of just `/mashing`, the page is `/skills/mashing`. Remember: the fact that our Contentful content instantly has URLs on our site comes from the Contentful package we installed earlier. But that magic has *nothing* to do with Layouts. So, customizing this URL is *also* specific to Contentful, not Layouts. But... I really want to fix it.

Creating the Slugger Class

Over in the `src/Layouts/` directory, create a new class called `ContentfulSlugger`. Make this implement `EntrySluggerInterface`... and then generate the one method we need: `getSlug()`:

src/Layouts/ContentfulSlugger.php

```
// ... lines 1 - 2
namespace App\Layouts;

use Netgen\Layouts\Contentful\Entity\ContentfulEntry;
use Netgen\Layouts\Contentful\Routing\EntrySluggerInterface;

class ContentfulSlugger implements EntrySluggerInterface
{
    public function getSlug(ContentfulEntry $contentfulEntry): string
    {
        // TODO: Implement getSlug() method.
    }
}
```

We're going to set things up so that this method is called when the dynamic URLs for *all* Contentful entries are being created. It will allow *us* to control the "slug", which is really the URL for each item.

To make life easier, use `FilterSlugTrait` to get access to a method we'll use in a minute:

src/Layouts/ContentfulSlugger.php

```
// ... lines 1 - 5
use Netgen\Layouts\Contentful\Routing\EntrySlugger\FilterSlugTrait;
// ... lines 7 - 8
class ContentfulSlugger implements EntrySluggerInterface
{
    use FilterSlugTrait;
// ... lines 12 - 20
}
```

Ok, on Contentful, we have both Skills and Advertisements. But we don't really want advertisements to have their own page. Unfortunately, with the Contentful integration, there's no way to *disable* URLs for one specific content type. I'll talk about how to work around that in a minute.

Anyways, this method will be passed both skills *and* advertisements. Use the new PHP `match()` function to match `$contentfulEntry->getContentType()->getId()`. That will return the internal name for each type, which you can find in Contentful. If it's `skill`, return `/skills/` then `$this->filtersSlug()` - that comes from the trait - passing `$contentfulEntry->get('title')`:

src/Layouts/ContentfulSlugger.php

```
// ... lines 1 - 8
class ContentfulSlugger implements EntrySluggerInterface
{
// ... lines 11 - 12
    public function getSlug(ContentfulEntry $contentfulEntry): string
    {
        return match ($contentfulEntry->getContentType()->getId()) {
            'skill' => '/skills/'.$this->filterSlug($contentfulEntry->get('title')),
// ... lines 17 - 18
        };
    }
}
```

For `advertisement`, return `/_ad` for all of them:

src/Layouts/ContentfulSlugger.php

```
// ... lines 1 - 8
class ContentfulSlugger implements EntrySluggerInterface
{
// ... lines 11 - 12
    public function getSlug(ContentfulEntry $contentfulEntry): string
    {
        return match ($contentfulEntry->getContentType()->getId()) {
            'skill' => '/skills/'.$this->filterSlug($contentfulEntry->get('title')),
            'advertisement' => '/_ad',
// ... line 18
        };
    }
}
```

At least, at *this* point, only *one* ad could ever have a page on our site: if the user went to `/_ad`, it would match the first one.

At the bottom, throw a new Exception with "Invalid Type":

src/Layouts/ContentfulSlugger.php

```
// ... lines 1 - 8
class ContentfulSlugger implements EntrySluggerInterface
{
// ... lines 11 - 12
    public function getSlug(ContentfulEntry $contentfulEntry): string
    {
        return match ($contentfulEntry->getContentType()->getId()) {
            'skill' => '/skills/'.$this->filterSlug($contentfulEntry->get('title')),
            'advertisement' => '/_ad',
            default => throw new \Exception('Invalid type'),
        };
    }
}
```

So, yes, at this point, advertisements *will* still have their own page. There's no way to turn that off out-of-the-box. But if you care enough, I would map all advertisements to the same URL or URL pattern like this. Then I would create a route & controller with the *same* URL and return a 404. That route will take precedence over the dynamic one.

Tagging & Configuring the Slugger

To tell Contentful to use our slugger, we need to, of course, give it tag! Add `#[AutoconfigureTag]` and this one is called `netgen_layouts.contentful.entry_slugger`. This also needs a `type` option... which you can set to *any* string. Let's use `default_slugger`:

src/Layouts/ContentfulSlugger.php

```
// ... lines 1 - 7
use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;

#[AutoconfigureTag('netgen_layouts.contentful.entry_slugger', ['type' =>
'default_slugger'])]
class ContentfulSlugger implements EntrySluggerInterface
{
// ... lines 13 - 22
}
```

How is that used? In `config/packages/`, we need to create a new config file for the layouts contentful package. Let's call it `netgen_layouts_contentful.yaml`.

Repeat that for the root key. Below, add `entry_slug_type`, then `default` set to the type we used in our tag: `default_slugger`:

config/packages/netgen_layouts_contentful.yaml

```
netgen_layouts_contentful:
    entry_slug_type:
        default: default_slugger
```

This funny syntax says:

> *"For every content type in Contentful, use `default_slugger` when generating the URL. So, use our `ContentfulSlugger`."*

Ok, done! But... this is *not* called when we reload the page. Nope. This is called when we "sync" our content from Contentful. Ok, let's re-sync! At your terminal, run:

```
symfony console contentful:sync
```

This updates our local database with the latest data from Contentful... and it worked just fine. But when we run:

```
symfony console contentful:routes
```

The URLs didn't change! This is a quirk... or maybe a feature so that existing pages don't break. Either way, once a route is imported the first time, it's URL never changes.

The easiest way to reset things is to drop the routes table and reimport everything.

And, this is kind of fun. We can run:

```
symfony console doctrine:migrations:migrate current-1
```

That will *reverse* the most recent migration, causing the contentful and route tables to be *dropped*. Put them back with:

```
symfony console doctrine:migrations:migrate
```

Re-sync the content again:

```
symfony console contentful:sync
```

And *now* check the routes:

```
symfony console contentful:routes
```

Yes! The URL is `/skills/mashing`! So, over on `/mashing`, we get a good-old fashioned 404. But `/skills/mashing` works.

Next: we don't yet have a page that lists *all* of the skills. Let's fix that!

Chapter 18: The Skills List Page + A Grid of Skills

Thanks to the Contentful integration, in addition to our `doctrine_recipe` value type, we now have a *second* value type that can load things from Contentful. This means that we can render lists and grids of *skills* inside any layout, like over here on our homepage.

Let's try it! Publish this layout... then edit the Homepage Layout. Oh, and we can delete this old grid we were playing with earlier.

Below, we're currently rendering the `featured_skills` Twig block. But in reality, if you looked at our template, those are totally hardcoded!

Adding A Grid of Skills

No problem! Add a Grid block... which is already set it to a "Manual Collection". But check this out! We can now choose between manually selecting "Contentful entries" or recipes! And when we click "Add Items", the content browser already works!

Select a few of these... good... then publish this. Refresh. Um... ok! They *do* render... but just the title. Good start. To make this a *tiny* bit better, go to the "Design" tab... and wrap this in a container.

That should, at least, give us some gutters. There we go. Ultimately, we want these to render like the hardcoded skills below them. And we're going to work on that in a few minutes.

Adding a /skills Page

But before we get there, what about a `/skills` page that lists *all* of the skills? Well, the Contentful integration did *not* give us this URL. But, no problem! We can create it ourselves in Symfony!

Well, actually, we *could* do this entirely in Contentful! We could create a "Page" content type, create a "Skills" page, which could become `/skills`, *then* map that to a Layout. This is the type of thing you'd normally do when you have a CMS at your fingertips

But we'll create this page the manual way. After all, Layouts is really about helping organize how *existing* pages look... it's not really about adding *dynamic* pages. That's a job for a CMS.

In your editor, open up `src/Controller/MainController.php`. Copy the `homepage()` action, paste, change to `/skills`, call it `app_skills` and rename the *method* to `skills()`. For the template, render `main/skills.html.twig`:

src/Controller/MainController.php

```
↕ // ... lines 1 - 8
9  class MainController extends AbstractController
10 {
↕ // ... lines 11 - 17
18     #[Route('/skills', name: 'app_skills')]
19     public function skills(): Response
20     {
21         return $this->render('main/skills.html.twig');
22     }
23 }
```

Now, in the `templates/main/` directory, create that: `skills.html.twig`. Let's start with the *smallest* possible thing: extend `nglayouts.layoutTemplate`:

templates/main/skills.html.twig

```
1 {% extends nglayouts.layoutTemplate %}
```

Cool. While we're here, open `base.html.twig` and link to this. Search for "Skills". There's the link. Set the `href` to `{{ path('app_skills') }}`:

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 18
    <body>
    {% block layout %}
        {% block navigation %}
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
// ... lines 23 - 32
            <div class="collapse navbar-collapse" id="navbarSupportedContent">
                <ul class="navbar-nav mr-auto">
// ... lines 35 - 37
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('app_skills') }}">All Skills</a>
                    </li>
                </ul>
// ... lines 42 - 47
            </div>
        </nav>
        {% endblock %}
// ... lines 51 - 67
    {% endblock %}
    </body>
</html>
```

I like it! Refresh, try the link in the header and... the page works!

Adding Content Manually?

To put *content* onto this page, we could *also* do that manually by writing code in our app! The Contentful library we installed earlier has a `ClientInterface` service that we could use to fetch all of these skills from Contentful in our controller.

But instead, let's take the easy way out and let *layouts* fetch the skills *for* us. Oh, but before we do that, back in `skills.html.twig`, add a `{% block title %}`, write "All Skills" and then `{% endblock %}`:

templates/main/skills.html.twig

```
{% extends nglayouts.layoutTemplate %}

{% block title %}All Skills{% endblock %}
```

This, as you probably know, controls the page's title. I'm doing this *here* because the `title` block is actually *not* something you can control via Layouts. Remember: everything we build in our layout becomes part of a *block* called `layout`.

Adding the Skill List Layout

Ok, hit "Publish" on the Homepage Layout... and then create a *new* layout. I'll use my favorite "Layout 2" and call it "Skills List Layout".

You know the drill. Start by linking the header zone... and the footer zone. Then, let's build another hero. Add a column, give it a `hero-wrapper` class, then put a "Title" block inside with "All Skills". To be even cooler, add a text block below with some intro content.

Nice start! Publish the layout... so we can go link it to the `/skills` page. Hit "Add New Mapping" and link this to the "Skills List Layout". Then go to Details. This time I'll map via the Path Info, set to `/skills`. Hit save changes.

Let's go see how our first attempt looks. And... not bad!

Adding the Skills Grid

Now let's add the *important* stuff. Head back to the layouts admin and edit this layout.

Below the column, add a new Grid. Change this from a manual collection to a dynamic collection. The Contentful package gives us *two* new "query types", or ways to "fetch" data from Contentful. Use "Contentful Search". That's the main one.

This allows you to choose *which* content types to show, like all of them... or just skills. We can then sort them, add a search, skip items or limit them. It's *everything* we want, out-of-the-box!

What does it look like? Hit "Publish". I bet you can guess. Yup! It "works"... by printing out the title of each skill. Oh, let me at least add that "container" class... to get the left and right margin.

But, this is *obviously* not what we want! We need to be able to *style* this and print out more fields than just the title. We have the same problem on the homepage.

And actually, this is even *more* complex than it seems! When we customize how a grid of skills renders, I want to be able to make those items look *one* way on the homepage, and a *different*

way on the "Skills" page, probably larger and with more fields printed.

Next: let's start learning the very important topic of how we can override and customize the templates from Layouts so that we can make things look exactly like we want.

Chapter 19: Themes & Overriding Templates

We can now add *a lot* of dynamic content to our site, like these static blocks up here, grids, or lists. The grids and lists can hold items from Contentful *or* our `Recipe` entity. But to *really* make our site shine, we need flexibility over *how* these pieces look. Let's start simple, by overriding the template that renders what the "Title" block looks like for our *entire* app.

Finding Block Templates in the Profiler

To do that... we *first* need to figure out *which* template is currently responsible for rendering this block. An easy way to find out is to go to a page that renders one of these, refresh, and click on the Twig icon on the web debug toolbar. Down at the bottom, we see the whole tree. And if we look closely, ah ha! Apparently there's a template called `block/title.html.twig`!

Layouts itself *also* has a really nice web debug toolbar section. If you go to "Rendered blocks", it shows "Block definition: title", "Text", "List", and "Footer". And, as we saw, the Title is rendered by `title.html.twig`.

Hello Themes

Notice that almost all of these templates are nestled inside `themes/standard/` directories. Layouts has a concept of *themes*, though we won't need to create *multiple* themes unless we're building some sort of multi-site application. In our case, we're just going to use the one *built-in* theme called `standard`.

But themes *are* still important, because anything *inside* of a theme can be easily overridden by putting a template in *just* the right location. We're going to use that convention to override the Title template.

Overriding the Title Template

Let's do it! First, in the `templates/` directory, make sure you have an `nglayouts/` subdirectory. Inside of that, add a new one called `themes/`... followed by a *another* subdirectory called `standard/`. You may have noticed that we're matching the structure that's over here: `nglayouts/themes/standard/`.

Inside of *this*, since the target template is named `block/title.html.twig`, if we create that *same* path, *our* `title.html.twig` will win. Do it: add another directory called `block/` and a new file inside: `title.html.twig`. To see if it works, just write some dummy text:

templates/nglayouts/themes/standard/block/title.html.twig

```
1  OVERRIDING TITLES!!
```

Let's try this thing! Go back to the Skills page, refresh, and... absolutely *nothing* happens. That's because the first time we create this `themes/` directory, we need to clear the cache.

```
php bin/console cache:clear
```

Do that... then with that behind us, try the page again. Woohoo! *We* now control how the Title block renders! The power!

Making the Title Template More Realistic

Okay, but even if we want to customize how the Title renders... we probably *don't* want to start from scratch. It would be better to *reuse* part of the core template, or at least use it as a reference.

Hit `Shift`+`Shift`, search for `title.html.twig`, and select "Include non-project items". Open the core one from `nglayouts/themes/`.

Wow. There is *a lot* going on here... including the fact that this extends another template: `block.html.twig`. Open that up.

This contains a *lot* of base functionality, like reading the dynamic `css_class` variable, which contains any CSS classes we enter into the admin. It also handles if there's a container or not. That's useful stuff!

In `title.html.twig`, it has code for whether or not the title is a link and other stuff. We *could* totally replace this template and ignore all this if we wanted to. But instead, copy the core template, paste it into our version:

templates/nglayouts/themes/standard/block/title.html.twig

```
{% extends '@nglayouts/block/block.html.twig' %}

{% import '@NetgenLayouts/parts/macros.html.twig' as macros %}

{% set tag = block.parameter('tag').value|default('h1') %}
{% set link = block.parameter('link') %}

{% block content %}
    {# Located inside the "content" block to include the context from the parent template #}
    {% set title = macros.inline_template(block.parameter('title').value, _context) %}

    <{{ tag }} class="title">
        {% if block.parameter('use_link').value and not link.empty %}
            {{ nglayouts_render_parameter(link, {content: title}) }}
        {% else %}
            {{ title }}
        {% endif %}
    </{{ tag }}>
{% endblock %}
```

And just to prove that we can, let's remove that `title` class:

templates/nglayouts/themes/standard/block/title.html.twig

```
// ... lines 1 - 7
{% block content %}
// ... lines 9 - 11
    <{{ tag }}>
// ... lines 13 - 17
    </{{ tag }}>
{% endblock %}
```

Cool! Now go over, refresh and... it goes back to how it looked before. But down here, that `title` class on the `<h1>` is gone!

So the simplest way to control how something looks is to find the template that renders it and override it completely using this `themes/` directory structure.

Let's use that trick *again* next to customize what it looks like when you render an "asset" field from Contentful, like this skill image field. But along the way, we're going to deep dive into a some massively important concepts: *block views* and *view types*.

Chapter 20: Block Views & View Types

Let's override one other template completely. Go into the Individual Skill Layout. We're using a Contentful entry here, which is a "Referenced asset"... and it's rendering as this image tag. Cool!

Block "View Types" / Templates

This is a great example of how a single Block type - for example the "Contentful Entry Field" block type - can have multiple *View types*, which basically means "multiple templates". Each of these different View types will be rendered by a different template. We actually see this with a lot of different Block types - even the Grid Block type. I'll add one down here temporarily. It has a View type that allows you to switch *between* List and Grid. Yup, the List and Grid blocks are *actually* both the *same* Block type internally: they just have a different View type, meaning each is rendered by a different template. Go ahead and delete that.

Anyway, every Block type can have one or more View types. And I actually want to dive a *little* deeper into this concept of "views". Find your terminal and run:

```
php ./bin/console debug:config netgen_layouts view
```

I'm debugging the configuration that could live under the `view` key below the `netgen_layouts` key:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 12
13     view:
↕  // ... lines 14 - 36
```

When you run this, you see a *ton* of config. Notice that there are several root keys, like `parameter_view`, `layout_view`, and a few others. But there are actually only *two* that we care about: `block_view`, which we'll talk about now, and `item_view`, which controls how the items

in a List or Grid render. We actually saw this one earlier when we customized how our Recipe "item" rendered inside a List or Grid. We'll talk even *more* about those soon.

The Block View Config

Anyways, to zoom in on the block views, run that same command, but add `.block_view`

```
php ./bin/console debug:config netgen_layouts view.block_view
```

Block views, very simply, control how *entire* block types are rendered. For example, we can see how the "Title block" renders... or the "Text block", *or* how the "List block" renders.

This `block_view` config can have several keys below it, like `default`, `app`, and `ajax`. And *we* know what those mean. `default` means these are used on the frontend, `app` means they're used in the admin section and `ajax`, which is *not* as common, is used on the frontend for AJAX calls. So to override the frontend template for a block, we really mean that we want to override its block "view" under the `default` key.

Let's... zoom in one more time by adding `.default`:

```
php ./bin/console debug:config netgen_layouts view.block_view.default
```

The "match" config

These are all the block views that will be used on the frontend. The *trickiest* thing about these are the `match` part.

When you define a "block view", it's pretty common to define the template that should be used when *two* things match. Search for "list\grid": this is a great example. This has *two* `match` items: `block\definition` is set to `list` because, *technically* the "Block type" for both the List and Grid blocks is called `list`. The second match condition is `block\view_type` set to `grid`.

Together these mean that if a block is being rendered whose `block\definition` is `list` and whose `block\view_type` is `grid`, use this.

By the way, both of these things can be seen very clearly from the web debug toolbar. Go to the homepage, click on the Layouts web debug toolbar, and go to "Rendered blocks". Down here... look at this! You can see "Block definition: List", "View type: grid"! And then it points to the template that was rendered. In this case, it's referring to this Grid right here.

So then... why is the Title block rendered by `title.html.twig`? We can see that in the config. Search for "title"... here we go. This says: if the `block\definition` is `title` and the `block\view_type` is `title`, use this template. This is an example of a Block type that only has *one* View type. So, in practice, this is the view that's used for *all* title blocks.

Find & Overriding the Contentful Field Assets View

Ok, let's remember our original goal: to override the template that renders this image. We know that this is a "Contentful entry field" and it has a View type of "Referenced assets". So... we can find that in here!

Search for "assets" and... there it is! So if `block\definition` is `contentful_entry_field` and the `block\view_type` is `assets`, this is the template! This means that if we want to override *just* the `assets` View type of the Contentful entry, *that's* the template we need to override.

And yes, we could have very easily found this by going to the web debug toolbar and finding the template *there*. But *now* we understand a bit more about how blocks are rendered and how each block can have multiple views so that we can *choose* how they're rendered. Later, we'll add an extra "view type" to an existing block.

Okay, so let's get to work. The path starts with the normal `nglayouts/themes/standard/`, *then* we need `block/`, followed by this path. So inside of our `block/` directory, create a *new* sub-directory called `contentful_entry_field/`. And inside of *that*, a new `assets.html.twig`. For now, I'll just say `ASSET`:

templates/nglayouts/themes/standard/block/contentful_entry_field/assets.html.twig

```
1 ASSET
```

Ok! Spin over to the frontend and... yes! It *instantly* sees it! We're now in control!

Making the Template Fancier

Like before, we probably don't want to override the *entire* template. Instead, open the core template - `assets.html.twig` - so we can steal, um borrow from it. Temporarily, copy the whole thing, paste:

templates/nglayouts/themes/standard/block/contentful_entry_field/assets.html.twig

```
{% extends '@nglayouts/block/block.html.twig' %}

{% block content %}
    {% set field_identifier = block.parameter('field_identifier').value %}
    {% set field = block.dynamicParameter('field') %}
    {{ dump() }}

    {% block contentful_entry_field %}
        {% if field is not empty %}
            {% if field.type is constant('TYPE_OBJECT', field) or field.type is constant('TYPE_ASSET', field) %}
                <div class="field field-{{ field.type }} field-{{ field_identifier }}">
                    <img src="{{ field.value.file.url }}?h={{ block.parameter('height').value }}&w={{ block.parameter('width').value }}" width="{{ block.parameter('width').value }}" height="{{ block.parameter('height').value }}" />
                </div>
            {% elseif field.type is constant('TYPE_ASSETS', field) %}
                <div class="field field-{{ field.type }} field-{{ field_identifier }}">
                {% for asset in field.value %}
                    <img src="{{ asset.file.url }}?h={{ block.parameter('height').value }}&w={{ block.parameter('width').value }}" width="{{ block.parameter('width').value }}" height="{{ block.parameter('height').value }}" />
                {% endfor %}
                </div>
            {% else %}
                {{ 'contentful.field_not_compatible'|trans({'%field_identifier%': field_identifier}, 'contentful') }}
            {% endif %}
        {% endif %}
    {% endblock %}
{% endblock %}
```

And... yep! That works.

Contentful is fairly advanced... and you can see that this supports fields that hold a single image as well as multiple images. You can keep this as flexible as you want, but you can *also* make it your own. I'm going to *drastically* simplify this template... and replace it with a very simple image. For the `src`, I'll paste in some code:

templates/nglayouts/themes/standard/block/contentful_entry_field/assets.html.twig

```
1 {% extends '@nglayouts/block/block.html.twig' %}
2
3 {% block content %}
4     {% set field = block.dynamicParameter('field') %}
5     {{ dump() }}
6     <img src="{{ field.value.file.url }}?h={{ block.parameter('height').value }}&w={{ block.parameter('width').value }}" />
7 {% endblock %}
```

All of the fancy Twig parts of this code were in the template before. This also shows off a Contentful superpower where you can control the image size. Calling `block.parameter()` allows us to read the *options* from the layouts admin, where we earlier configured this block to have a width and height of 200.

Let's see what it looks like! Refresh. Yeah! It looks like it worked!

Choosing to Render or Not Render Complex Options

But I *do* want to want give one small warning about customize templates: make sure you don't lose flexibility that you need. For example, we know that we can add extra CSS classes to any block via the admin.

If we did that *right now*, it would *not* work because... we're simply not rendering those classes! And, that might be fine. But if you *do* want to support that, you'll need to make sure to add it. In this case we can say `class="{{ css_class }}"`, which is one of the variables we saw earlier. And while we're here, let's also add an `alt` attribute set to `field.value.title`:

templates/nglayouts/themes/standard/block/contentful_entry_field/assets.html.twig

```
↕ // ... lines 1 - 2
3 {% block content %}
↕ // ... line 4
5     <img class="{{ css_class }}" src="{{ field.value.file.url }}?h={{ block.parameter('height').value }}&w={{ block.parameter('width').value }}" alt="{{ field.value.title }}" />
6 {% endblock %}
```

When we try this... I love it! There's the `alt` attribute and *there's* our class, including some core classes that Layouts always adds to that variable.

Okay, we just talked about block views: how templates are configured for entire blocks. Next, let's talk about *item views*: how we customize the template that's used when rendering an *item* inside of a Grid or List. We'll use this to *style* our skill items.

Chapter 21: Deep Dive into Item Views

When it comes to customization, you can do *a lot* of damage by looking at which templates are rendering and using the theme system to override them. *But* there *are* a few cases where you'll need to get even more specific.

For example, suppose we want to modify the "item" template for how the skills grid renders on the homepage. If you check the web debug toolbar here and scroll down... I'll actually search for "contentful"... ah, there we go. You can see `grid.html.twig`... which renders `item/contentful_entry.html.twig`. To customize the item, we *could* override that template. *Easy peasy*.

The *problem* is that, in Contentful, we have *multiple* content types: we have Skills *and* Advertisements. So if we override this template, that will override it for *both* Skills and Advertisements... and we *probably* want those to look different. So, how can we solve this?

Diving into the item_view Config

Earlier, we ran `debug:config netgen_layouts view` and talked about the two main sections under here - `block_view` (which controls how blocks render) and `item_view`.

```
php ./bin/console debug:config netgen_layouts view.item_view
```

As I've said a few times, *some* blocks, like Grid and List, render individual *items*. This `item_view` config is where we define *those* templates. And you'll see some familiar root keys: `default` for the frontend, `ajax` for AJAX calls, and `app` for the admin. Once again, this uses the `match` config and... hey! We see our entry in here! Remember `recipes_default`? We configured this inside of our config file, and it's one of the two *real* item templates we have right now:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
                # this key is not important
                recipes_default:
                    template: 'nglayouts/frontend/recipe_item.html.twig'
                    match:
                        item\value_type: 'doctrine_recipe'
// ... lines 29 - 36
```

There's one for recipes, and then Contentful has one for *all* of the Contentful items.

So again, we *could* just override this template via the themeing system and be done. But our goal is to override this template *only* when the item is a skill, like this one. So how do we do that? By adding our *own* `item_view` to this list that matches that *single* content type. Let's do it!

Adding a Custom item_view

Over here... we're under `item_view`, `default` for the frontend and we have the one entry from earlier: `recipes_default`. Let's add another. Call it `contentful_entry/skill`, though this particular key doesn't make any difference. Below that, set `template` to `@nglayouts/item/contentful_entry`, followed by `skill.html.twig`:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
// ... lines 24 - 28
                contentful_entry/skill:
                    template: '@nglayouts/item/contentful_entry/skill.html.twig'
// ... lines 31 - 41
```

Before, we were using `nglayouts` *without* the `@`... just because I told you that `nglayouts/` was a nice directory for organizing things. Internally, Layouts uses `@nglayouts` in its template paths. What's the difference? By adding the `@`, we're hooking into the themeing system. So because we have a `templates/nglayouts/` directory with `themes/standard/` inside, it will use our template. Feel free to use `@nglayouts` or just `nglayouts`. I just wanted you to understand the difference because you'll see the `@nglayouts` syntax all over the place.

Matching just One Content Type

The *really* important key here is `match`. We want to match *only* when we're working with a `contentful_entry`. Ok, copy `match` from the config... and paste.

But we need to be more specific. We *also* need to match only when the *type* of the content is a *skill*. But how do we do that? What matchers are even available? There *is* a core list... but did Contentful add any *additional* matchers that we could leverage?

Here's a little trick to see the *true* list of `match` items. It's a *little* technical, but works beautifully. Run:

```
php ./bin/console debug:container --tag=netgen_layouts.view_matcher
```

What is this doing? Well, anyone can create a *custom* matcher - like `foo\bar`. To *do* that, you create a class and give it this tag. By looking for all services *with* that tag, we can find *all* of the existing matchers in the system.

And... look at that list! Oh, here's an interesting one: `contentful\content_type`. I bet we can use that. Try it: `contentful\content_type` set to `skill`:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
// ... lines 24 - 28
                contentful_entry/skill:
                    template: '@nglayouts/item/contentful_entry/skill.html.twig'
                    match:
                        item\value_type: 'contentful_entry'
                        contentful\content_type: 'skill'
// ... lines 34 - 41
```

Okay, let's go create the template. Inside `themes/standard/`, instead of `block/`, this time, create a directory called `item/`... then `contentful_entry/`, and *then* `skill.html.twig`. Just put some dummy text for now:

templates/nglayouts/themes/standard/item/contentful_entry/skill.html.twig

```
CONTENTFUL SKILL!
```

Ok, *if* this is working, when we refresh, these items - which are Contentful skills - *should* re-render using our new template. But when we try it... absolutely *nothing* changes. What happened?

Wrong Config Order!

Go back to your terminal and run

```
php ./bin/console debug:config netgen_layouts view.item_view
```

again. This all looks good... except for the *order*. This one from Contentful is on the *top* of the list... and our new ones are at the bottom. And guess what? When Layouts tries to figure out which template to render, it reads the list from top to bottom and finds the *first* one that matches: exactly how Symfony's routing system works.

So, it first looks at `contentful_entry`, sees that the `value_type` *is* `contentful_entry`... then uses it. It *never* makes it to the `contentful_entry/skill` at the bottom.

To fix this, we're going to use a fancy *prepend* configuration trick. Let's do that *next*.

Chapter 22: Prepending Config

I'm *pretty* sure that our new `item_view` is configured correctly. We have `item\value_type: contentful_entry`, which I *know* is correct... and then we're using `contentful\content_type` set to `skill` so that this *only* affects skills:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
// ... lines 24 - 28
                contentful_entry/skill:
                    template: '@nglayouts/item/contentful_entry/skill.html.twig'
                    match:
                        item\value_type: 'contentful_entry'
                        contentful\content_type: 'skill'
// ... lines 34 - 41
```

But... it doesn't seem to be working on the frontend. Earlier, when we ran `debug:config`, we saw that the *problem* lies with the *order* of the config. Layouts reads from top to bottom when deciding which "view" to use. So it looks at this one first, sees that the `value_type` *is* `contentful_entry`... and just stops. To fix this, we need to *reverse* our config.

Ok, so... why is it in this order to begin with? Why does our config show up at the bottom? This is due to how Symfony loads config: it loads *bundle* config *first* - like from the Contentful package or Layouts - and *then* loads *our* configuration files. And, that's *usually* the order we want! It allows us to *override* configuration that's set in bundles.

But in *this* case, we want the opposite. How do we accomplish that? By asking Symfony to *prepend* our configuration.

Setting up the Prepend

In the `config/` directory, create a new directory called `prepends/` and move the Netgen Layouts configuration into it. This will stop Symfony from loading that file in the normal way: we're going to load it manually.

The next step is a bit technical. In `src/`, create an "extension" class called, how about, `AppExtension`. I'm going to paste in the code: you can grab this from the code block on this page:

src/AppExtension.php

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

use Symfony\Component\Config\Resource\FileResource;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Extension\Extension;
use Symfony\Component\DependencyInjection\Extension\PrependExtensionInterface;
use Symfony\Component\Yaml\Yaml;

class AppExtension extends Extension implements PrependExtensionInterface
{
    public function load(array $configs, ContainerBuilder $container)
    {
    }

    public function prepend(ContainerBuilder $container)
    {
        $configFile = __DIR__ . '/../config/prepends/netgen_layouts.yaml';
        $config = Yaml::parse((string) file_get_contents($configFile));
        $container->prependExtensionConfig('netgen_layouts', $config['netgen_layouts']);
        $container->addResource(new FileResource($configFile));
    }
}
```

This loads our config file like normal... except that it will be *prepended*.

Final step. To get this method to be called, open up the `Kernel` class. After `use MicroKernelTrait`, add `configureContainer as baseConfigureContainer`:

src/Kernel.php

```
// ... lines 1 - 10
class Kernel extends BaseKernel
{
    use MicroKernelTrait { configureContainer as baseConfigureContainer; }
// ... lines 14 - 20
}
```

This adds the `configureContainer` method from `MicroKernelTrait` *into* this class like a trait normally would... except that it *renames* it to `baseConfigureContainer`. We're doing this so that we can define our *own* `configureContainer()` method. Copy the `configureContainer()` signature from the trait, paste, hit "OK" to add the `use` statements and then call `$this->baseConfigureContainer()` passing `$container`, `$loader`, and `$builder`:

src/Kernel.php

```
// ... lines 1 - 5
use Symfony\Component\Config\Loader\LoaderInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Loader\Configurator\ContainerConfigurator;
// ... lines 9 - 10
class Kernel extends BaseKernel
{
// ... lines 13 - 14
    private function configureContainer(ContainerConfigurator $container, LoaderInterface $loader, ContainerBuilder $builder): void
    {
        $this->baseConfigureContainer($container, $loader, $builder);
// ... lines 18 - 19
    }
}
```

The `configureContainer()` method in the trait is responsible for loading `services.yaml` as well as all of files inside `config/packages/`. That's all good stuff that we want to *keep* doing. But after doing that, add one more thing: `$builder->registerExtension(new AppExtension())`:

src/Kernel.php

```
// ... lines 1 - 10
class Kernel extends BaseKernel
{
// ... lines 13 - 14
    private function configureContainer(ContainerConfigurator $container, LoaderInterface $loader, ContainerBuilder $builder): void
    {
        $this->baseConfigureContainer($container, $loader, $builder);

        $builder->registerExtension(new AppExtension());
    }
}
```

Again, yes, this is annoyingly technical. But thanks to these two pieces, our `netgen_layouts.yaml` config will be *prepended*.

Check it out! Re-run the `debug:config` command again:

```
php ./bin/console debug:config netgen_layouts view.item_view
```

Scroll up and... yes! Our configuration is *now* on top! And when we refresh... woohoo! We see the text!

Next: let's make this template render *exactly* like the hardcoded skills. Then we'll create a *second* item template to customize how the Contentful "Advertisement" content type renders.

Chapter 23: Contentful Item Template

Our "item" template for skills *is* now being used! So, let's finish it!

We already know what we want the skills to look like... so let's go steal that from `templates/main/homepage.html.twig`. Find the `featured_skills` block, copy what *one* of those skills looks like, and paste that into `skill.html.twig`. Let's also add `dump(item.object)` at the top:

templates/nglayouts/themes/standard/item/contentful_entry/skill.html.twig

```
1  {{ dump(item.object) }}
2  <a href="#" class="text-center skill-item-container p-3">
3      <h3>Folding in Cheese</h3>
4      <div class="p-3 mt-3 skill-img">
5          <img src="{{ asset('images/pup-cup.png') }}" width="auto" height="80"
   alt="grater skill image">
6      </div>
7  </a>
```

We've created an item template before, so we know `item.object` *should* give us the underlying "object" that represents this Contentful entry.

If we head over and refresh... *awesome*! This dumps a `ContentfulEntry` object. And, though you can't see it from here, this class has a `get()` method we can use to read *any* of the underlying data from Contentful.

For skills, if we dig a little... we have fields like `title` and `shortDescription`. Let's use those! For example, in the `<h3>`, say `{{ item.object.get('title') }}`:

templates/nglayouts/themes/standard/item/contentful_entry/skill.html.twig

```
1  {{ dump(item.object) }}
2  <a href="#" class="text-center skill-item-container p-3">
3      <h3>{{ item.object.get('title') }}</h3>
↕  // ... lines 4 - 6
7  </a>
```

And... yup! That worked!

For the `<img src="">`, replace the `asset()` stuff with `item.object.get('image')`, followed by `.file.url`, which is specific to Contentful. Also fill in the `alt` attribute with `item.object.get('title')`:

templates/nglayouts/themes/standard/item/contentful_entry/skill.html.twig

```
<a href="{{ path('cmf_routing_object', {'_route_object': item.object}) }}"
class="text-center skill-item-container p-3">
    <h3>{{ item.object.get('title') }}</h3>
    <div class="p-3 mt-3 skill-img">
        <img src="{{ item.object.get('image').file.url }}" width="auto"
height="80" alt="{{ item.object.get('title')}} skill image">
    </div>
</a>
```

The *last* thing we need to update is the URL. But, hmm. If we had created a "skill show" page in Symfony, we could use the Twig `path()` function to link to that route! However, each skill page is *actually* created via a dynamic route thanks to the Contentful bundle. And, to create those routes, it uses something called the CMF routing system.

So, to link, we need to use that system. Say `path('cmf_routing_object')` and pass `_route_object` set to `item.object`:

templates/nglayouts/themes/standard/item/contentful_entry/skill.html.twig

```
<a href="{{ path('cmf_routing_object', {'_route_object': item.object}) }}"
class="text-center skill-item-container p-3">
// ... lines 2 - 5
</a>
```

If you were using Sylius or Ibexa CMS, you would use some function from *their* system to create the link: this is specific to the CMF routing system.

Head over and try that. Yes! And if we click the link... double yes!

Let's celebrate by removing the `dump()`. We can also delete this `featured_skills` block from our homepage template: We won't need that at all anymore. But let's remake this `<h2>` inside of the layouts admin first. To do that, add a Title block called "Featured Skills", make that "H2"... and give it those same CSS classes: `text-center mb-4`.

The Grid is already in a container... but we want *all* of this in a container. So add a Column, wrap *that* in a Container, move the Grid and Title blocks inside of it... then we won't need a Container on the Grid anymore. Delete the "Features Skills" block... then finally hit "Publish and continue editing". While we're waiting, delete that block also from the Twig template.

And now... yes! It looks perfect!

The Advertisement Item View

Okay, while we're talking about item views, let's customize the item template for our *other* content type inside of Contentful: *Advertisement*. We're only rendering that in *one* place, on the individual skill page... right over *here*. Let's go check that out.

Publish this layout... then edit the *individual* skill layout. Earlier, we used the Contentful Entry Field block to render the `advertisement` field, which is a "referenced entity". Yup, if you modify a skill in Contentful, down on the bottom, the "Advertisement" field allows you to choose from the Advertisements in our system.

Click on the Twig icon of the web debug toolbar... search for "item", and scroll down.. No surprise: it's using the standard Contentful "item" template. And, good news, we already know how to override that!

Head over to our configuration, copy the `contentful_entry/skill` section, and paste it below. Replace `skill` with `ad` for the section name and `template`... and update the `content_type` to `advertisement`... because that's the internal name of that type in Contentful:

config/prepends/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    view:
        item_view:
// ... lines 15 - 21
            # default = frontend
            default:
// ... lines 24 - 33
                contentful_entry/ad:
                    template: '@nglayouts/item/contentful_entry/ad.html.twig'
                    match:
                        item\value_type: 'contentful_entry'
                        contentful\content_type: 'advertisement'
// ... lines 39 - 46
```

Ok! Let's go add that template. In `contentful_entry`, create a new file called `ad.html.twig`... and then just print some text: `Advertisement`:

templates/nglayouts/themes/standard/item/contentful_entry/ad.html.twig

```
Advertisement
```

Moment of truth. Head back over and refresh. We got it! That was easy!

For the *real* template contents, I'll just paste them in. Once again, we use `item.object.get()` to read the `url` field. There's also an `image` field and a `shortText` field:

templates/nglayouts/themes/standard/item/contentful_entry/ad.html.twig

```
<a href="{{ item.object.get('url') }}" class="p-3 text-center ad-item-container" target="_blank">
    <h3>Sponsored Product</h3>
    <img src="{{ item.object.get('image').file.url }}" width="auto" height="200" alt="Ad image">
    <p class="pt-3">{{ item.object.get('shortText') }}</p>
</a>
```

And now... *we've got it*!

Next: What if we wanted to create a Grid of items... but make that *one* Grid look different than *every* other Grid on the site? We can do that by creating an extra "block view" for an existing block.

Chapter 24: Block Views & Block Definitions

Let's create a layout for our "individual recipe" page so that we can customize this a bit more. I love that we can create new layouts on the fly, whenever a page needs to be tweaked.

Adding and Mapping a New Layout

Add a new layout, choose our favorite Layout 2 and call it "Individual Recipe Layout". And y'all know the drill at this point. Start by linking the Header zone... then the Footer zone.

Cool! And then because we're going to be applying this to a normal page that already exists in Twig, add a "Full View" block, which will render the `body` block from our template.

Solid start. Hit "Publish"... so we can map this. Add a new mapping, link it to the "Individual Recipe Layout"... then hit "Details". This time, let's link via the route name. To get that name, open `src/Controller/RecipeController.php`. Here it is: `app_recipes_show`. Paste that, hit "Save Changes" and... let's try this!

We shouldn't see any difference yet and... we don't. But we *can* see that it's using our layout!

Let's spice this page up a bit! Go back to the layouts admin and edit the "Individual Recipe Layout". Add a new Grid and change it to a "Dynamic collection"... that uses "Contentful search". Load Skills, show the newest first and limit to 3.

Ok, if we "Publish and continue editing"... then refresh... whoa! It's cool that we can just put those anywhere now. Though, let's wrap that in a container. And... much better.

So far, this is all easy! Ready for the complication? I want to customize how this grid looks: I want to have one *big* recipe on the left and then two smaller recipes on the right. But I do *not* want to change how the grid looks on *other* parts of our site, like on the homepage. So the question is: how can we change how this grid renders on *just* this page?

The Grid/List View Types

Click on the Grid and go to design tab. It turns out that a Grid is really just a "List" block. And the "List" block has two "view types": list and grid.

Head over to your terminal and run:

```
php ./bin/console debug:config netgen_layouts view.block_view
```

Oh, but spell `netgen` correctly. This displays the configuration for how blocks are rendered. Find the `default` section... then scroll down a bit. Here: we see the two view types for list and grid. As I mentioned, it turns out that these are actually both part of the *same* block *type* called `list`. They're just two different *view types*: one called `list` and one called `grid`. When you switch the "view type" in the layouts admin, you're effectively switching which *template* is used to render that block.

Block Definitions

Run that same command, but instead of `view.block_views`, check `block_definitions`:

```
php bin/console debug:config netgen_layouts block_definitions
```

Block definitions is where you define what the blocks actually *are*. So every root key under this config represents a different block that we can use inside the admin area. Find the one called `list`: here it is. This defines things like what form fields are rendered in the admin area for this block *and* what "view types" it has. This has two: list and grid. Layouts reads *this* config to render the "View Type" `select` field in the admin. Then, once we select the view type, it uses the `block_views` config we looked at before to know which *template* to render.

Ok, enough deep config and theory. Let's give ourselves a *new* way to render lists by creating a new view type. That's next.

Chapter 25: Custom Block View

So here's the plan. We're going to add a new "view type" to the list block definition. Then we're going to map that to a *template* via `block_views`.

Updating the "Block Definition"

For step 1, open our `netgen_layouts.yaml` file and, really anywhere, add `block_definitions`. This config can be used to create totally *new* blocks or change options on *existing* blocks, which is what *we* want. To do that, we need to repeat the config here: `list` & `view_types`. So, `list` `view_types` and then add the new one. Let's call it `one_by_two` - that key can be anything - and give it a name: `1x2 Featured Grid`:

config/prepends/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    block_definitions:
        list:
            view_types:
                one_by_two:
                    name: 1x2 Featured Grid
// ... lines 18 - 52
```

Just by doing that, if we go over and refresh the admin area... and click down on the grid, we have a new view type! If we change to it... nothing renders in the admin area. And if we hit "Publish and continue editing"... over on the frontend... *also* nothing renders. Yay!

Click the Layouts link in the web toolbar and... near the bottom, ah. It's rendering `invalid_block.html.twig`. The block definition *is* `list` and the view type *is* `1x2 Featured Grid`. The problem is that we haven't, yet, defined a "block view" for that combination. So, it falls back to "invalid block".

Adding the Admin Block View

Ok, under `view`, we've already created several "item views". *Now* add `block_view` so we can create our first of *those*. We're going to register both an admin view as well as a frontend view. Because... in the admin area, it currently renders nothing. Add `app` for the admin and the next key doesn't matter. For the template, because the admin view isn't too important, let's re-use the core admin "grid" template, which you could find via the `debug:config` command. It's `@NetgenLayoutsStandard/app/block/list/grid.html.twig`.

Now add `match`. We want to use this template if `block\definition` is `list` *and* `block\view_type` is `one_by_two`... making sure that this matches the key we used earlier under the block definition:

config/prepends/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 18
    view:
// ... lines 20 - 52
        block_view:
            app:
                list/one_by_two:
                    template:
'@NetgenLayoutsStandard/app/block/list/grid.html.twig'
                    match:
                        block\definition: list
                        block\view_type: one_by_two
// ... lines 60 - 67
```

How did I know to use `block\definition` and `block\view_type`? By using our favorite `debug:config` command! That's always a good guide to follow.

Anyways, that should fix the admin area. And... it does!

Frontend Block View

For the frontend view, duplicate that entire section... but use `default`. This key is fine, it doesn't matter, and change the template to, how about, `@nglayouts/block/list/one_by_two_list.html.twig`. The match section is perfect already:

config/prepends/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 18
    view:
// ... lines 20 - 52
        block_view:
// ... lines 54 - 60
            default:
                list/one_by_two:
                    template: '@nglayouts/block/list/one_by_two_list.html.twig'
                    match:
                        block\definition: list
                        block\view_type: one_by_two
```

Ok, let's go make that template! We already have `templates/nglayouts/themes/standard/block/`... so, create the new `list` subdirectory then the file: `one_by_two_list.html.twig`. Start by saying `1x2`:

templates/nglayouts/themes/standard/block/list/one_by_two_list.html.twig

```
1x2
```

Let's check it! Over on the frontend, refresh and... there's our tiny 1x2!

Customizing the Frontend Template

Let's bring this to life! Because this renders a "list" block, our template probably has access to some variable that represents the "items". To cheat, which is *always* a good choice for developers, let's peek at the core grid template: `grid.html.twig` from the `themes/` directory.

Wow! Like many core templates, there's a lot of stuff in here! You can choose what you want to keep or get rid of. The most important thing is this `collection_html` variable: they loop over `collections[collection_identifier]`... where `collection_identifier` is actually just the word `default`. So it loops over `collections.default`. Then it includes a template. That `templateName` variable will be set to something like `grid/` the number of columns `.html.twig`. For example, if the grid is configured to use 3 columns, it would use `3_columns.html.twig`. That template adds the `div` needed for each column in a 3 column setup... and then calls `nglayouts_render_result()`. *That* renders the "item".

Anyways, if you zoom out, the template basically loops over the `collections` variable and calls `nglayouts_render_result()` on each one.

Back in our template, I'm going to paste in some code that does something similar:

templates/nglayouts/themes/standard/block/list/one_by_two_list.html.twig

```
{% extends '@nglayouts/block/block.html.twig' %}

{% block content %}
    <div class="row">
        {% for result in collections.default %}
            <div class="col-sm-6 col-md-6 col-lg-4">
                {{ nglayouts_render_result(result, null, block.itemViewType) }}
            </div>
        {% endfor %}
    </div>
{% endblock %}
```

Yup, we extend `block.html.twig`, just like the core template does, then loop over `collections.default`, add a `div` and render each item. So this is effectively a simpler version of what a grid does.

And what does it look like? Refresh and... yup! It looks like a grid!

But remember the goal: one big skill on the left with two smaller skills on the right. To make that happen, I'll paste in version 2 of my template. Nothing special here. Instead of looping, this renders the 0 key, then the 1 and 2 keys:

templates/nglayouts/themes/standard/block/list/one_by_two_list.html.twig

```
{% extends '@nglayouts/block/block.html.twig' %}

{% block content %}
    <div class="row">
        <div class="col-6">
            {{ nglayouts_render_result(collections.default[0], null, block.itemViewType) }}
        </div>

        <div class="col-6">
            <div class="row">
                <div class="col-6">
                    {{ nglayouts_render_result(collections.default[1], null, block.itemViewType) }}
                </div>
                <div class="col-6">
                    {{ nglayouts_render_result(collections.default[2], null, block.itemViewType) }}
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

And now... yes! That's *exactly* what I wanted!

Though, I'll give you the same warning I gave earlier when we were overriding core "item" templates. We are *not* including all of the custom stuff that lives in the core template. If you need to support a custom option, make sure to include that code.

Hiding Block Options for a Block View Type

And actually, one thing in here - the number of columns - is *not* something we need. This is something that we can configure for the block... but it's not relevant at *all* when using our new view type.

Could we... hide that option when using our view type? Yep! Head back to your terminal and debug the `block_definitions` config again:

```
php ./bin/console debug:config netgen_layouts block_definitions
```

Search for `one_by_two`. We *could* configure this `valid_parameters` key to *remove* an option from the block. The `list` view type does exactly that. I won't do it, but that's how it's done.

Ok, head back to the site and go to the "All Skills" page. Yea... things still don't look right. On this layout, we're using a grid to render the items. And, that grid looks ok on *other* pages but not here, where the skills are meant to be the *main* content on the page. Next, let's learn how we can customize how these *items* render for *just* this grid.

Chapter 26: Custom Item View Type

The Grid of skills on the `/skills` page looks *terrible*. Let's go find the layout for that: Skills List Layout. Ok, so this is a *normal* Grid... and it renders like any *other* Grid on the site. I want to *customize* this, but I don't want the Grid block *itself* to render differently: having them tiled like this is great. What I *really* want is to change how each *item* inside the grid is rendered... but *just* in this one situation. How can we do that?

Hello "Item View Types"

Head over to your terminal and run our favorite `debug:config` command, this time looking at `block_definitions`:

```
php ./bin/console debug:config netgen_layouts block_definitions
```

This is, as we learned, the *config* for all of the blocks in our system. And check this out! One piece of config we haven't talked about yet is `item_view_types`. For each "block view type", like `one_by_two`, `list`, or `grid`, there's also `item_view_types`. So far, *all* of these currently have a single one called `standard`.

It's not *super* common, but for a given view type - like `one_by_two` or `list` - you can specify *multiple* ways to render the *items* inside of that view type. Those are called `item_view_types`. `Standard` is the default, and it means the items will render in the "normal" way.

So here's our goal: for the existing `grid` view type, we're going to add a *new* "item view type". On a high level, this will allow us, while configuring a grid, to choose a *different* way to render the items.

To start, over in our configuration, find `block_definitions`. We currently have `list`, `view_types`, and `one_by_two`. *Now* add `grid` so we can override that existing view type. Add `item_view_types` with a new one called, how about, `skill_big_view`. You'll see how we use that key in a second. Also give this a human-readable name:

config/prepends/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 12
    block_definitions:
        list:
            view_types:
// ... lines 16 - 17
                grid:
                    item_view_types:
                        skill_big_view:
                            name: Skills Big View
// ... lines 22 - 77
```

What did that do? Refresh the admin area... click down on the Grid... and make sure you're on the "Design" tab. Hey! We have a new "Item view type" select! It shows "Standard", which is the default, then our new "Skills Big View"!

Select it and hit "Publish and continue editing". What will this change on the frontend when we refresh? Absolutely nothing! That's because we now need a new "item view" that will *match* this.

Adding the "Item View" for the new "Item View Type".

Back in our config, scroll down to `item_views`. Below `default`, copy the `contentful_entry/skill` section and paste it *above*.

We're putting it *above* because order is important: we need this *new* item view to be able to match *before* the other one. Watch. Call this `contentful_entry/skill_big_view` and change the template to `@nglayouts/item/contentful_entry/skill_big_view.html.twig`. We *still* want to match when `item\value_type` is `contentful_entry` and `contentful\content_type` is `skill`... but *only* if the matcher called `item\view_type` equals the key we created earlier `skill_big_view`:

config/prepends/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 22
    view:
        item_view:
// ... lines 25 - 31
            # default = frontend
            default:
// ... lines 34 - 38
                contentful_entry/skill_big_view:
                    template:
'@nglayouts/item/contentful_entry/skill_big_view.html.twig'
                    match:
                        item\value_type: 'contentful_entry'
                        contentful\content_type: 'skill'
                        item\view_type: 'skill_big_view'
// ... lines 45 - 77
```

Thanks to this, if the user selects this as their "Item View Type" for a grid of skills, then all *three* of these will match. But if the user chooses the default `Standard` item view type, it would *not* match this... but it still *would* match the one below.

Let's go add the template. Inside `item/contentful_entry/`, create the new file: `skill_big_view.html.twig`. Inside, say `BIG VIEW`:

templates/nglayouts/themes/standard/item/contentful_entry/skill_big_view.html.twig

```
BIG VIEW
```

Let's try it! Make sure the layout is published... then on the frontend... we got it! The rest is easy! Because we've already created several item view templates... I'll just paste in the rest. There's nothing new here.

But now... yea! *That* is the look we're going for.

Changing the "Item View" on an Item by Item Basis

By the way, now that our "Grid" block view has multiple "item view types" - that's our configuration up here - we have the power, on an item-by-item basis, to control that. See this "Override slot view type"? This basically says;

> *“Yo Layouts! I want to change whatever the first item in this list is to use the "Standard" view type.”*

I'll hit "Publish and continue editing" and now... you can see that just the *first* item uses the Standard view type! That's... obviously not what we want on our site, so I'll go back and use "No overrides". But *that* is a very powerful concept.

And... woh! Just one chapter left! One common problem with Layouts is working with vertical spacing: just making sure the spacing is correct between all of our components. We *could* control that by adding CSS classes to individual blocks. But wouldn't it be nice if *every* block in the system had a nice drop down where we could select the top and bottom margins automatically? How can we make a modification to an *existing* block, or even *all* blocks in our app? That's the job of a *block plugin*, and that's *next*.

Chapter 27: Block Plugins

Well look at us! We've made it to the *last* topic of the tutorial. We've already transformed our static site into one where we can reorder the layout for each page, mix it with custom code from Twig templates and add dynamic content. That's... kind of awesome. Of course, we haven't covered everything you can do with Layouts, but you're now truly dangerous.

Creating a Custom Block?

One topic that we *haven't* covered is how to create a totally *new* block, but this *is* documented and, at this point, I think it wouldn't be too hard. Why *would* you build a custom block? Suppose you have something super custom like our "Hero" area or this "subscribe to newsletter" area, which is actually powered by Symfony's UX Live Component package, which gives it the fancy Ajax behavior.

Anyways, if you want something like this on your page, the simplest way to add it is... how *I* did in this project: put the logic in Symfony, render inside a Twig block, then *include* that Twig block inside of Layouts.

But what if we want the admin user to be able to add this to *multiple* pages whenever they want? *That* is when creating a custom block would be useful. Custom blocks can also have options, so you could even let them customize this in some way.

Hello Block Plugins

Anyways, let's do one last challenge related to blocks: create a block plugin. Go to a skill show page. Hmm, we could probably use a bit more margin between these blocks. And that's a pretty common need. We *could* handle this by adding a CSS class that sets the margin. But I want to make it even *easier*.

Go to the Layouts admin, then edit the Individual Skill Layout. Ok, suppose we want to add some margin right here. To do that, I want the admin user to be able to click on *any* block in the

system - for example, this column block - and over on the design tab, select the top or bottom margin they need from a new form field.

This is a pretty wild goal... because, to accomplish it, we need to be able to *modify every* block in the system! Fortunately, *that* is exactly the point of a block plugin: to *extend* one - or every - block.

Creating the Block Plugin

Let's get to work. In the `src/Layouts/` directory, create a new PHP class called, how about, `VerticalWhitespacePlugin`. This needs to implement a `PluginInterface`. But in practice, we extend a `Plugin` class that implements that interface for us. Go to "Code"->"Generate", or `Command`+`N` on a Mac, and implement the one method we need: `getExtendedHandlers()`:

src/Layouts/VerticalWhitespacePlugin.php

```
// ... lines 1 - 2
namespace App\Layouts;

use Netgen\Layouts\Block\BlockDefinition\Handler\Plugin;

class VerticalWhitespacePlugin extends Plugin
{
    public static function getExtendedHandlers(): iterable
    {
        // TODO: Implement getExtendedHandlers() method.
    }
}
```

Ok, each block in the system - so every item over here on the left menu - has a *class* behind it called a block handler. Our job in `getExtendedHandlers()` is to return an iterable of *all* the "handlers" that we want to extend. For example, if you wanted to *only* extend the title block, you could `yield TitleHandler::class`. How did I know to use that class? Well, most of the time you can guess: the title block has a `TitleHandler`. But if you want to look deeper, you can see *all* the handlers in the system by running:

```
php bin/console debug:container --tag=netgen_layouts.block_definition_handler
```

Anyways, in our case, we want to override *every* block. So we can `yield BlockHandlerDefinitionInterface::class`, because *every* block handler must implement that interface:

src/Layouts/VerticalWhitespacePlugin.php

```
// ... lines 1 - 4
use Netgen\Layouts\Block\BlockDefinition\BlockDefinitionHandlerInterface;
// ... lines 6 - 7
class VerticalWhitespacePlugin extends Plugin
{
    public static function getExtendedHandlers(): iterable
    {
        yield BlockDefinitionHandlerInterface::class;
    }
}
```

And yes, I *totally* just forgot the word `Definition`. Whoops! I'll fix this bad interface in a minute.

Adding a Custom Block Parameter/Field

To see what to do next, go back to the "Code"->"Generate" menu, select "Override methods" and choose `buildParameters()`. We don't need to call the parent method because it's empty:

src/Layouts/VerticalWhitespacePlugin.php

```
// ... lines 1 - 6
use Netgen\Layouts\Parameters\ParameterBuilderInterface;
// ... lines 8 - 9
class VerticalWhitespacePlugin extends Plugin
{
// ... lines 12 - 16
    public function buildParameters(ParameterBuilderInterface $builder): void
    {
// ... lines 19 - 27
    }
}
```

Parameter is the word that Layouts uses for the form *options* that you can customize on the right side of the screen for every block. Thanks to our `getExtendedHandlers()` method, when Layouts builds those options for *any* block, it will now call *this* method and we can add *new* parameters.

I'll paste in the first... and we also need a `use` statement for this `ParameterType` namespace:

src/Layouts/VerticalWhitespacePlugin.php

```
// ... lines 1 - 7
use Netgen\Layouts\Parameters\ParameterType;

class VerticalWhitespacePlugin extends Plugin
{
// ... lines 12 - 16
    public function buildParameters(ParameterBuilderInterface $builder): void
    {
        $builder->add(
            'vertical_whitespace:enabled',
            ParameterType\Compound\BooleanType::class,
            [
                'default_value' => false,
                'label' => 'Enable Vertical Whitespace?',
                'groups' => [self::GROUP_DESIGN],
            ],
        );
    }
}
```

Cool! As you can see, Layouts comes with a bunch of built-in "field types" - like `BooleanField`, which will render as a checkbox. It defaults to false and has a label. Oh, and this group? Remember how there are two tabs - "Design" and "Content"? This is where you determine which your parameter should live inside.

And the first key - `vertical_whitespace:enabled` is the internal name of this field. You'll see how we use that in a minute.

Before we try this, future Ryan has just informed me that... I messed up! Typical. Scroll up. I'm yielding the wrong class! Yield `BlockDefinitionHandlerInterface::class`:

src/Layouts/VerticalWhitespacePlugin.php

```
// ... lines 1 - 4
use Netgen\Layouts\Block\BlockDefinition\BlockDefinitionHandlerInterface;
// ... lines 6 - 9
class VerticalWhitespacePlugin extends Plugin
{
    public static function getExtendedHandlers(): iterable
    {
        yield BlockDefinitionHandlerInterface::class;
    }
// ... lines 16 - 28
}
```

That's better.

Now let's try it. Refresh... click on any block... let me find my Title block... and... there it is! On *any* block we see the new field!

Adding "Child" Parameters/Fields

But, the *real* idea is that, if the user enables this, we show them two *more* fields where they can select the top or bottom margin.

To do that, after the first field, I'll paste in two more parameters:

src/Layouts/VerticalWhitespacePlugin.php

```
// ... lines 1 - 9
class VerticalWhitespacePlugin extends Plugin
{
// ... lines 12 - 16
    public function buildParameters(ParameterBuilderInterface $builder): void
    {
        $builder->add(
            'vertical_whitespace:enabled',
// ... lines 21 - 26
        );

        $builder->get('vertical_whitespace:enabled')->add(
            'vertical_whitespace:top',
            ParameterType\ChoiceType::class,
            [
                'default_value' => 'medium',
                'label' => 'Top Spacing',
                'options' => [
                    'None' => 'none',
                    'Small' => 'small',
                    'Medium' => 'medium',
                    'Large' => 'large',
                ],
                'groups' => [self::GROUP_DESIGN],
            ],
        );

        $builder->get('vertical_whitespace:enabled')->add(
            'vertical_whitespace:bottom',
            ParameterType\ChoiceType::class,
            [
                'default_value' => 'medium',
                'label' => 'Bottom Spacing',
                'options' => [
                    'None' => 'none',
                    'Small' => 'small',
                    'Medium' => 'medium',
                    'Large' => 'large',
                ],
                'groups' => [self::GROUP_DESIGN],
            ],
        );
    }
}
```

These are basically like the first. The big difference is that, up here, we said `$builder->add()`. But *now* we have `$builder->get('vertical_whitespace:enabled')` and *then* `->add()`. This makes these *child* fields under the first.

This is pretty cool. Refresh and... let's find the Column block. Click to "Enable Vertical Whitespace". Woh! The other two fields showed up! Let's do "Medium" top spacing and "No" bottom spacing. Publish that.

Using the Parameters in the Block Template

It shouldn't be *too* surprising, however, that when we refresh the page... absolutely nothing happens! We added those options... but we're not *using* them anywhere yet. We need to override a template to do that.

Let's think: we want this top and bottom margin to apply to *every* block in the system. And, fortunately, every block in the system eventually extends `block.html.twig`: this one here in the `nglayouts/themes/` directory.

Copy this. Then override it via the theming system. If we follow the path... `standard/block`... `standard/block`... the new file should live here: `block.html.twig`. Paste the contents inside.

To make sure this is working, put a little `TEST`:

templates/nglayouts/themes/standard/block/block.html.twig

```
{% set css_class = ['ngl-block', 'ngl-' ~ block.definition.identifier, 'ngl-vt-' ~ block.viewType, css_class|default(block.parameter('css_class').value)]|join(' ') %}
{% set css_id = css_id|default(block.parameter('css_id').value) %}
{% set set_container = block.parameter('set_container').value %}

{% if show_empty_wrapper is not defined %}
    {% set show_empty_wrapper = false %}
{% endif %}

{% set block_content = (block('content') is defined ? block('content') : '')|trim %}

{% if block_content is not empty or show_empty_wrapper %}
    <div class="{{ css_class }}" {% if css_id is not empty %} id="{{ css_id }}" {% endif %}>
        TEST
        {% if set_container %}<div class="container">{% endif %}

        {{ block_content|raw }}

        {% if set_container %}</div>{% endif %}
    </div>
{% endif %}
```

Ok! Refresh the frontend. Yikes! Yep, that's definitely working. Go... take that out.

At the top of the template, we have a variable called `css_class`, which is set to some core classes. And hey! It calls `block.parameter('css_class')`! Yup, *that's* what reads the "CSS class" field from the block options!

Then, it uses `|join(' ')` to combine all of these into a string.

I'm going to *remove* that `join()`... then rename this variable to `css_classes`:

templates/nglayouts/themes/standard/block/block.html.twig

```
{% set css_classes = ['ngl-block', 'ngl-' ~ block.definition.identifier, 'ngl-vt-' ~ block.viewType, css_class|default(block.parameter('css_class').value)] %}
↕ // ... lines 2 - 21
```

We're setting things up so that we can easily *modify* that variable. Down here, right before `block_content`, recreate that `css_class` variable set to `css_classes|join(' ')`:

templates/nglayouts/themes/standard/block/block.html.twig

```
// ... lines 1 - 8
{% set css_class = css_classes|join(' ') %}
{% set block_content = (block('content') is defined ? block('content') : '')|trim %}
// ... lines 11 - 21
```

This variable is used in a bunch of different places *and* in child templates. So we need to make sure it's still set.

Anyways, up here, we now have a `css_classes` array. Let's use that! I'll paste in three variables, each set to the value of our three parameters:

templates/nglayouts/themes/standard/block/block.html.twig

```
// ... lines 1 - 2
{% set set_container = block.parameter('set_container').value %}

{% set use_whitespace = block.parameter('vertical_whitespace:enabled').value is same as(true) %}
{% set whitespace_top = block.parameter('vertical_whitespace:top').value %}
{% set whitespace_bottom = block.parameter('vertical_whitespace:bottom').value %}
// ... lines 8 - 29
```

This is where the parameter name we used in the class comes in handy.

Now, very simply, if `use_whitespace`, then add some margin classes. I'll paste that code in too:

templates/nglayouts/themes/standard/block/block.html.twig

```
// ... lines 1 - 4
{% set use_whitespace = block.parameter('vertical_whitespace:enabled').value is same as(true) %}
{% set whitespace_top = block.parameter('vertical_whitespace:top').value %}
{% set whitespace_bottom = block.parameter('vertical_whitespace:bottom').value %}
{% if use_whitespace %}
    {% set css_classes = css_classes|merge(['whitespace-top-' ~ whitespace_top]) %}
    {% set css_classes = css_classes|merge(['whitespace-bottom-' ~ whitespace_bottom]) %}
{% endif %}
// ... lines 12 - 29
```

So, for the top margin, we're adding a new `whitespace-top-` followed by `none`, `small`, `medium` or `large`. And same for the bottom.

These new classes are totally invented: they're not part of Bootstrap CSS or anything else, but you *could* make this smarter to reuse those. But for us, if you open `assets/styles/app.css`... near the top, here we go!

assets/styles/app.css

```
// ... lines 1 - 12
.whitespace-top-small {
    padding-top: 2rem;
}
.whitespace-top-medium {
    padding-top: 4rem;
}
.whitespace-top-large {
    padding-top: 8rem;
}
.whitespace-bottom-small {
    padding-bottom: 2rem;
}
.whitespace-bottom-medium {
    padding-bottom: 4rem;
}
.whitespace-bottom-large {
    padding-bottom: 8rem;
}
// ... lines 31 - 108
```

Before the tutorial, I already prepared those classes.

So... it should work! Move over and refresh. Got it! Our block has a little extra top whitespace... which comes from our new class.

And... done!, Woo! Great job team! You're now a Layouts champion! Let us know what cool things you're building with it. And if you have any questions, as always, we're here for you down in the comments section.

Alright, thank you and seeya next time.

With <3 from SymfonyCasts