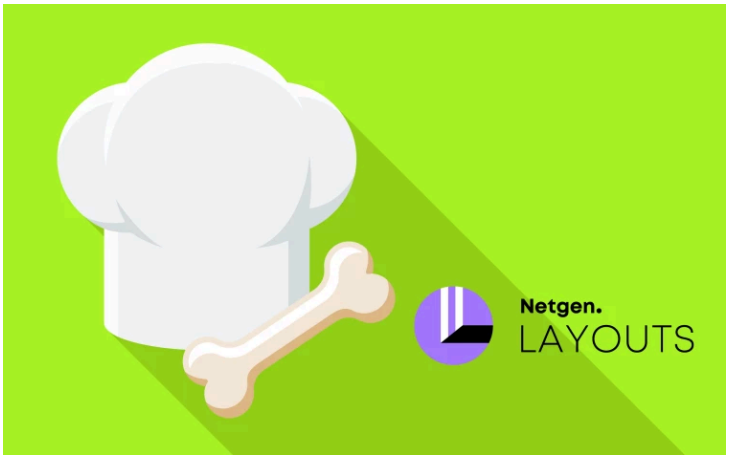
Netgen Layouts: Construyendo páginas con Symfony



Chapter 1: ¡Hola Configuración de Layouts+!

¡Hola amigos! Me alegro mucho de que estéis aquí conmigo, porque este tutorial trata de algo divertido, genial y bastante poderoso. No, no se trata de un felino enmascarado que lucha contra el crimen y que tiene superpoderes, aunque eso sería genial. Este tutorial trata sobre el paquete Netgen Layouts.

¿Qué es Layouts?

Esta biblioteca existe desde hace años, pero hace poco que la he probado. Layouts es, sencillamente, una forma de tomar cualquier aplicación Symfony existente y añadir la capacidad de reorganizar dinámicamente la organización de tus páginas sobre la marcha a través de una sección de administración... incluyendo la adición de nuevo contenido dinámico. Es una mezcla muy interesante de una aplicación Symfony normal con controladores y plantillas Twig... además de funciones de gestión de contenidos que puedes activar página por página. A mí me gusta especialmente el enfoque de la opción

¿Quién necesita diseños?

¿Por qué te tomarías la molestia de usar Layouts en tu aplicación Symfony? Bueno, no todos los proyectos lo necesitan. Pero si un usuario administrador necesita poder hacer cambios en la organización de tu sitio y su contenido, entonces esto es para ti. Esto incluye poder añadir y cambiar colecciones de elementos -como productos destacados- justo en el centro de una página existente, reorganizar el contenido de una plantilla Twig más arriba o más abajo en la página, añadir un contenido personalizable completamente nuevo a una página o crear páginas de destino temporales y permitir que todo esto lo hagan los usuarios normales. Puedes aprovechar Layouts para una sola página de tu sitio, dejando que todo lo demás sea una aplicación normal de Symfony, o todas las páginas de tu sitio pueden utilizarlo. Como he dicho, puedes optar por los Layouts como mejor te parezca.

Configuración del proyecto

Podría seguir y seguir, pero probablemente sea mejor ver la magia de Layouts en acción. Es súper divertido jugar con él, así que definitivamente deberías descargar el código del curso desde esta página y codificar junto a mí. Cuando descomprimas el archivo, encontrarás un directorio `start/` con el mismo código que ves aquí. Abre este archivo `README.md` para ver todos los detalles de la configuración. Yo ya he ido a mi terminal, he instalado mis activos Node mediante

```
yarn install
```

y he ejecutado:

```
yarn watch
```

para construir mis archivos CSS y JS. Pero todo eso es sólo para que nuestra aplicación y este tutorial sean más realistas. Layouts no requiere que usemos Encore y no se mete con nuestro CSS y JS en absoluto.

De todos modos, el último paso en el `README` es abrir otra pestaña del terminal y ejecutar

```
symfony serve -d
```

para iniciar un servidor web en https://127.0.0.1:8000 - Voy a hacer trampas y hacer clic en eso. Y... hola nuevo proyecto paralelo: ¡es Bark & Bake! Escucha, los perros están bastante cansados de nuestros perezosos intentos de cocina canina. ¿Problemas crujientes? No, gracias. Así que hemos construido este sitio para inspirar a la gente a ser los mejores chefs que puedan ser... para sus perros.

Se trata de una aplicación Symfony bastante tradicional con algunos controladores y algunas plantillas Twig. También tiene dos entidades: Una entidad `User` para la seguridad, y una entidad `Recipe`. En el sitio, tenemos una página de inicio que muestra las últimas y mejores recetas, una sección de recetas y la posibilidad de abrir una receta específica, para que podamos seguirla en la cocina. Eso es todo. Esto de las habilidades no está implementado en absoluto todavía.

Así que, aparte de poder editar los detalles de cada receta a través de un área de administración, ¡este es un sitio estático! ¡Es hora de cambiar eso! Pronto podremos tomar esta página de inicio -que está escrita mediante un controlador y una plantilla Symfony normales... como puedes ver aquí- y utilizar diseños para insertar contenido y reorganizar cosas de forma dinámica

Instalar los layouts

Así que vamos a instalar Layouts. Busca tu terminal y ejecuta:

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

Esto descargará varios paquetes, incluyendo un par de bundles nuevos. Cuando termine, ejecuta

```
git status
```

para ver que también nos ha proporcionado dos nuevos archivos de rutas, que añaden algunas rutas de administración que vamos a ver en unos minutos.

Ejecuta las migraciones

Layouts también requiere algunas tablas adicionales de la base de datos donde se almacena la información sobre los diseños que crearemos, así como cualquier contenido personalizado que vayamos a poner en ellos. Veremos todo eso en la sección de administración de Diseños en un minuto. Para añadir las tablas necesarias, desplázate hacia arriba y copia esta ingeniosa línea `doctrine:migrations:migrate`.

Esto es genial. Los paquetes de layouts vienen con migraciones... y esto las ejecuta. Pega este comando, pero si utilizas la configuración de la base de datos de Docker que describimos en el LÉEME - yo lo hago - entonces modifica esto para que empiece con `symfony console` para que pueda inyectar las variables de entorno de Docker que apuntan a nuestra base de datos:

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

Y... ¡perfecto! Una advertencia es que estas migraciones están escritas específicamente para MySQL. Por ahora, Layouts sólo admite MySQL.

Ignorar las tablas personalizadas

En su mayor parte, Layouts va a gestionar por completo todas las tablas que acabamos de añadir: no necesitamos hacer nada con ellas. Pero ahora que existen en nuestra base de datos, si añadiéramos una nueva entidad y generáramos una migración para ella... la migración intentaría eliminar todas las tablas de Netgen Layouts. ¡Observa! Ejecuta:

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

Esto imita la generación de una migración, y... ¡sí! Quiere soltar todas las tablas de Diseños. Para solucionarlo, entra en `config/packages/doctrine.yaml` y, en `dbal`, añade `schema_filter`, y pasa una expresión regular... que puedes copiar de la documentación de Layouts:

config/packages/doctrine.yaml

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

¡Perfecto! Con esto, si volvemos a ejecutar el comando `doctrine:schema:update` de nuevo...

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

Está limpio, excepto por `doctrine_migration_versions`. Pero, no te preocupes: el comando `make:migration` es lo suficientemente inteligente como para no dejar caer su propia tabla.

Bien, Netgen Layouts está instalado y tiene las tablas de la base de datos que necesita. Si volvemos a actualizar nuestro sitio ahora... ¡felicidades! No ha cambiado absolutamente nada, aunque tenemos un bonito icono de la barra de herramientas de depuración web aquí abajo del que hablaremos más adelante.

Esto, de nuevo, es una de las grandes cosas de Layouts. Al instalarlo no se apodera de tu aplicación. Layouts no se utiliza en absoluto para representar esta página.

A continuación, vamos a sumergirnos en el área de administración de Layouts para crear nuestro primer diseño. Luego, veremos cómo interactúa con las páginas reales de nuestro sitio.

Chapter 2: Crear y mapear diseños

Bien, veamos de qué se trata Layouts. En este capítulo, crearemos y utilizaremos, paso a paso, un "diseño", aprendiendo cómo funciona exactamente la magia de los Diseños.

Para comprobar la sección de administración de los Diseños, dirígete a `/nglayouts/admin` y encontrarás... ¡un formulario de acceso! El formulario de inicio de sesión no tiene nada que ver con los Layouts... es que el área de administración de los Layouts requiere que estés conectado... y ya he añadido un formulario de inicio de sesión a nuestro sitio. ¡Incluso hay un usuario en la base de datos! Inicia sesión con `doggo@barkbite.com`, contraseña `woof`.

El rol de seguridad necesario para el área de administración

Y cuando lo enviamos... ¡acceso denegado! No te preocupes: haz clic abajo en el icono de seguridad de la barra de herramientas de la web... y ve a "Decisión de acceso". Sí: nos han denegado el acceso porque buscaba un rol llamado `ROLE_NGLAYOUTS_ADMIN`. Para acceder al área de administración de los diseños, necesitamos tener este rol.

La forma más sencilla de añadirlo es ir a `config/packages/security.yaml`. El usuario con el que estamos conectados ahora mismo tiene `ROLE_ADMIN`. Por lo tanto, en `role_hierarchy` también le damos a nuestro usuario `ROLE_NGLAYOUTS_ADMIN`:

config/packages/security.yaml

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

Creando nuestro primer diseño

Y ahora si volvemos a hacer clic, ¡tachán! ¡Bienvenido a la sección de administración de layouts! Para entender lo que hace Layouts... lo mejor es verlo en acción. Empieza en esta sección de Diseños... y haz clic para crear un nuevo diseño. Esto nos muestra unos seis tipos

de diseño diferentes entre los que podemos elegir. Como verás, son mucho menos importantes de lo que puede parecer al principio, porque, una vez que estás en una disposición, puedes hacer realmente lo que quieras, incluso hacer flotar cosas a izquierda y derecha. Yo suelo elegir la "Disposición 2". Llámalo "Diseño de la página de inicio" porque pienso utilizarlo en nuestra página de inicio.

Y... ¡bienvenido al editor de diseño! Recorrido rápido: estos elementos del lado izquierdo se llaman "bloques", y hay muchos tipos diferentes, desde simples bloques de título a mapas de Google... hasta cosas más complejas como listas y cuadrículas en las que puedes representar colecciones dinámicas de cosas, como recetas destacadas. Lo principal que "hacemos" en esta página es elegir un bloque de la izquierda... y arrastrarlo a una de las "zonas" del centro.

Colocar los bloques en el diseño

Coge un bloque "Título" y arrástralo a algún lugar de la página... luego dale algo de texto. ¡Genial!

Es un comienzo modesto, pero, ¡bastante bueno! En la parte superior derecha, pulsa "Publicar diseño".

Y ahora que tenemos este nuevo diseño, abre una segunda pestaña y ve a la página de inicio para descubrir que... ¡no ha cambiado absolutamente nada! Deja que reorganice mis pestañas.

Mapeo de un diseño

De todos modos, nada ha cambiado porque, una vez que tienes una disposición, necesitas asignarla a una página o conjunto de páginas específicas. Ese es el trabajo de la sección de mapeo de diseños. Realmente son las dos únicas secciones importantes en el área de administración.

Aquí, añade un nuevo mapeo y luego ve a Detalles. Hay varias formas de asignar un diseño a una URL específica. Puedes utilizar, por ejemplo, la información de la ruta, que es un término elegante que significa "la URL, pero sin parámetros de consulta". O puedes usar un prefijo de información de la ruta, como usar esta disposición para todas las URL que empiecen por "/productos". O incluso puedes asignar un diseño a un nombre de ruta específico.

Vamos a probar eso. Pulsa "Añadir objetivo". Entonces... vamos a buscar el nombre de la ruta de nuestra página de inicio: `src/Controller/MainController.php`. Aquí está: `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
    }
}
```

Vuelve a desplazarte, pega y pulsa "Guardar objetivo".

Más adelante hablaremos de otras formas de mapear o "activar" un diseño para las páginas, pero la información de rutas y caminos es la más sencilla y flexible. Dicen:

> *"Si la ruta o la URL actuales coinciden con lo que tenemos aquí, utiliza este diseño."*

Pulsa guardar cambios. Para elegir qué maquetación va con este mapeo, pulsa "Maquetación de enlace" y selecciona la única: "Diseño de la página de inicio".

¡Genial! Ahora, cuando vayamos a la página de inicio, se utilizará el diseño de la página de inicio. Pero... ¿qué significa eso? ¡Vamos a averiguarlo! Actualiza y... ¡todavía no vemos ninguna diferencia! ¡Es la misma página estática de Symfony!

Ampliando el diseño base dinámico

Ah, eso es porque nos hemos saltado un paso importante de la instalación. ¡Culpa mía! Ve a abrir la plantilla de esta página: `templates/main/homepage.html.twig`. Ahora mismo, estamos ampliando `base.html.twig`:

templates/main/homepage.html.twig

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

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

Y esa plantilla, como es habitual, tiene un bloque llamado `body` en el centro:

templates/base.html.twig

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

Así que es una configuración súper tradicional.

Ahora, cambia el `extends` por una variable dinámica llamada `nglayouts.layoutTemplate`:

templates/main/homepage.html.twig

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

Configurar el diseño base

Prueba la página de nuevo. ¡Error! ¡Eso es un progreso! Dice:

> *"Diseño base de la página, no especificado. Para renderizar la página con Layouts, especifica el diseño base de la página con esta configuración."*

Todo esto tendrá más sentido dentro de un minuto. Lo que quiere que hagamos es abrir `config/packages/` y crear un nuevo archivo -que puede llamarse como sea- pero llamémoslo `netgen_layouts.yaml`. Dentro, añade `netgen_layouts` y, debajo, `pagelayout` ajustado a nuestro `base.html.twig`:

config/packages/netgen_layouts.yaml

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

Te explicaré todo esto en un minuto. Si refrescamos ahora... ¡eh, el mismo error! Es posible que Symfony no haya visto mi nuevo archivo de configuración... así que déjame borrar la caché para estar seguro:

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

Y ahora... ¡sí! ¡Funciona! Excepto que... ¡sigue siendo la misma página estática! Pero, por primera vez, abajo en la barra de herramientas de depuración de la web, muestra que se está utilizando el "Diseño de la página de inicio". Así que se ha dado cuenta de que el diseño debe ser utilizado... sólo que no parece estar renderizándolo.

Renderizar el bloque de diseño

Para solucionarlo, tenemos que hacer una última cosa... y luego retrocederemos para explicar lo que está pasando y lo genial que es. En `base.html.twig`, alrededor de `{% block body %}`, añade `{% block layout %}`... luego después de `{% 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>
```

Actualiza una vez más. Y... ¡guau! ¡Nuestra página ha desaparecido! Vale, todavía tenemos el nav y el pie de página... que vienen de arriba y de abajo de los bloques en `base.html.twig`, pero el contenido real de nuestra página ha desaparecido y ha sido sustituido por el bloque de título dinámico ¿Qué magia negra es ésta?

La magia de la herencia de plantillas de diseño

En primer lugar, antes de explicarlo, permíteme decir que hay formas mucho más rápidas de empezar con Netgen Layouts: tienen proyectos de inicio para aplicaciones normales de Symfony, aplicaciones de Sylius y aplicaciones de Ibexa CMS. Pero hemos hecho todo este

trabajo de configuración manualmente a propósito... porque realmente quiero que entiendas cómo funcionan los Layouts: es sorprendentemente sencillo.

En primer lugar, nuestra página sigue llegando a nuestra ruta normal - `app_homepage` - y sigue ejecutando nuestro controlador normal y sigue renderizando nuestra plantilla normal. No hay nada de magia ahí.

Pero entonces, extendemos `nglayouts.layoutTemplate`. ¿A qué apunta eso? Si no hay ningún diseño asignado a una página concreta, `nglayouts.layoutTemplate` resolverá a `base.html.twig`. Eso es gracias a la configuración que hemos añadido aquí:

config/packages/netgen_layouts.yaml

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

Pero si Layouts encuentra una asignación de diseño para esta página, entonces `nglayouts.layoutTemplate` resuelve a una plantilla de Layouts del núcleo. En este caso, si pulsas `Shift`+`Shift`, se llamará `layout2.html.twig`... ya que hemos seleccionado "Layout 2".

Esto renderiza el diseño dinámico a través de estas etiquetas `nglayouts_render_zone`: cada una de ellas se refiere a una sección diferente -o "zona"- dentro de nuestro diseño.

En cualquier caso, lo realmente importante es que convierte el diseño en un bloque Twig llamado `layout`. A continuación, extiende `ngLayouts.pageLayoutTemplate`, que resuelve a nuestro `base.html.twig`.

El resultado final es que nuestra página se renderiza con total normalidad y sigue extendiendo `base.html.twig`... pero se ha añadido un bloque llamado `layout` que contiene el contenido del diseño dinámico.

Por eso no vimos ningún cambio en la página al principio. Hasta que no incluimos `{% block layout %}` en `base.html.twig`, el diseño se estaba cargando... sólo que no lo estábamos mostrando en ningún sitio.

La conclusión es la siguiente: si estás en una página que no está maquetada, todo es exactamente igual que siempre. Pero si estás en una página que se mapea a un diseño, simplemente significa que ahora tienes un bloque llamado `layout` cuyo contenido es igual a lo que tengas dentro de ese diseño.

Extender el diseño dinámico a todas las páginas

Como he mencionado antes, no tenemos que añadir maquetas a todas las páginas de nuestro sitio: ¡podríamos añadirlas a la página de inicio y listo! Pero todas las páginas que queramos que admitan diseños tienen que ampliar `nglayouts.layoutTemplate`. Lo bueno es que, incluso si ampliamos esto, no ocurre nada a menos que asignemos un diseño a esta página. Así que no hay ningún inconveniente en utilizarla en todas partes. Actualizaré rápidamente `login.html.twig` para utilizarlo:

templates/security/login.html.twig

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

luego `list.html.twig` y `show.html.twig`:

templates/recipes/list.html.twig

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

templates/recipes/show.html.twig

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

¡Realmente puedo moverme rápido cuando lo necesito!

De vuelta al navegador, las páginas de la lista de recetas y de la muestra de recetas siguen teniendo el mismo aspecto... porque no se ha resuelto el diseño. Pero ahora están preparadas para usar maquetas, si queremos.

Ahora, por muy interesante que sea controlar dinámicamente el contenido de la página de inicio, ¡hemos hecho demasiado! Todo nuestro antiguo contenido ha desaparecido. ¿Es posible mezclar el contenido dinámico con parte del contenido estático de nuestra plantilla Twig de la página de inicio? Por supuesto. Y eso es una gran parte de lo que hace que los diseños sean especiales. Eso a continuación.

Chapter 3: Añadir bloques Twig a tu diseño dinámico

Acabamos de sustituir por completo nuestra página de inicio por un diseño dinámico. Pero, eso no es realmente tan interesante. Lo que realmente quiero es poder utilizar mi plantilla de página de inicio existente y todo este buen contenido que he preparado:

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 %}
```

pero luego retocarla añadiendo pequeños fragmentos de contenido dinámico aquí y allá... o incluso reorganizar las cosas. Para ello, en el diseño, debajo de los bloques, en la parte inferior, añade uno especial llamado "Bloque Twig"... y pongámoslo justo debajo del título. Fíjate en que puedes poner tantos bloques como quieras dentro de una misma zona. En realidad, estas zonas no acaban siendo tan importantes.

De todas formas, cuando hagas clic en un bloque, en la parte derecha verás las opciones de ese bloque, que tiene una importante llamada "Nombre del bloque Twig". Introduce `body` para que coincida con el `{% block body %}` que tenemos en la plantilla:

templates/main/homepage.html.twig

```
{% extends nglayouts.layoutTemplate %}

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

Vale, dale a "publicar y continuar editando"... luego ve a actualizar la página de inicio. ¡Santo hombre murciélago del contenido! Nuestro contenido Twig vive ahora dentro de esta página dinámica. ¡Es increíble! Y todo sigue funcionando: incluso el elegante "componente vivo" del centro de la página.

Añadir más bloques a tu plantilla

Vale, esto es genial... pero sigue siendo sólo un montón de contenido dinámico en la parte superior... y luego contenido de plantilla Twig en la parte inferior: realmente no podemos mezclar nada en el centro de nuestra página.

A menos que... añadamos más bloques a nuestra plantilla. Por ejemplo, mantener el `block body`... sólo para que la página siga funcionando aunque no mapeemos un diseño... pero luego añadir un `{% block hero %}` alrededor de la sección superior, un bloque llamado, qué tal, `latest_recipes`, `{% endblock %}`, otro llamado `subscribe_newsletter`, `{% endblock %}` y uno final llamado `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 %}
```

Si nos detuviéramos ahora, esto no supondría ninguna diferencia para nuestra app: seguimos renderizando el bloque `body` aquí abajo... que incluye todos esos. Pero acabamos de darnos un montón de poder nuevo.

Compruébalo: cambia el nombre del bloque `body` por `hero`. Y luego vamos a añadir unos cuantos bloques Twig más. Renderiza `latest_recipes` para éste. Por cierto, las "etiquetas" de los bloques son sólo para nosotros en el área de administración: sólo para mayor claridad. Si introduzco "Últimas recetas", aparecerá encima del bloque. Totalmente opcional.

Añade dos bloques más: uno que muestre `subscribe_newsletter` y, por último, uno para `featured_skills`. Luego, aquí arriba, voy a eliminar por ahora el bloque `title`.

Por cierto, estoy utilizando la palabra "bloque" para referirme a dos cosas distintas a la vez. Los bloques son las "cosas" que añadimos a nuestro diseño, como un título, un mapa de Google o

una lista de elementos. Pero los bloques también se refieren a los bloques Twig de nuestras plantillas. Y, por supuesto, uno de los tipos de bloques que podemos añadir... es uno que renderiza... bloques Twig. Bloques Twig. Un poco confuso, pero no puede ser peor.

De todos modos, di "Publicar y continuar editando"... y luego ve a actualizar el frontend. Y... ¡genial! Nuestra página funciona. Lo sé, tiene exactamente el mismo aspecto que hace un minuto, pero ahora está siendo renderizada por layouts... ¡y podemos reorganizar las piezas!

Observa: Moveré el `subscribe_newsletter` hacia abajo, le daré a "Publicar y continuar editando", actualizaré y... ¡boom! Esa parte estática de la página se ha movido mágicamente a la parte inferior. Es genial.

O podríamos volver a subirla... y luego añadir algún contenido dinámico, como texto, entre uno de los otros bloques.

A continuación, vamos a ser aún más agresivos y flexibles permitiendo que la navegación superior y el pie de página inferior sean opcionales, pero fáciles de añadir, dentro del Diseño.

Chapter 4: Diseños compartidos

Abre `base.html.twig` y mueve el `{% block layout %}` para que esté alrededor de todo. Así, pon el inicio justo dentro de la etiqueta `body`... y el final justo antes de la etiqueta de cierre `body`:

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>
```

Si ahora actualizamos la página de inicio... ¡se destruye! La parte superior `nav` y `footer` han desaparecido. ¿Por qué he hecho esto? ¡Porque me encanta el caos! Es broma, lo he hecho porque nos da el poder, dentro de los layouts, de diseñar páginas totalmente personalizadas: incluso páginas sin los tradicionales `navigation` y `footer`, tal vez como una página de aterrizaje temporal para una promoción.

Pero seamos sinceros, el 99% de las veces, querremos los `nav` y `footer`. No hay problema, vuelve a `base.html.twig`. Recuerda: añadir bloques nos da más flexibilidad. Así que, encima de la navegación, añade un nuevo bloque llamado `navigation`, con `{% endblock %}` después. Luego, aquí abajo, otro llamado `footer`... y `{% 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>
```

Apuesto a que sabes lo que voy a hacer a continuación. En el administrador del diseño, ahora podemos añadir un bloque Twig en la parte superior que muestre `navigation`... y otro aquí abajo en la parte inferior. No es necesario que esté en esta última zona... pero tiene sentido allí. Renderiza `footer`.

¡Vamos a probarlo! Pulsa "Publicar y continuar editando" y... actualiza. ¡Ya estamos de vuelta!

Crear una segunda maqueta

Vamos a crear un segundo diseño, esta vez para la página `/recipes`. Si miras en `RecipeController`, verás que ya he hecho todo el trabajo para consultar las recetas y pasarlas a esta plantilla:

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
}
```

Y en esa plantilla, hacemos un bucle y renderizamos cada una, con paginación:

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 %}
```

Y así, definitivamente quiero incluir todo este trabajo personalizado en el nuevo diseño.

De vuelta al área de administración, pulsaré "Publicar diseño" como una forma fácil de volver a la lista de diseños. A continuación, pulsa en nuevo diseño, elegiré mi diseño favorito 2 y lo llamaré "Diseño de la lista de recetas". Para empezar, añade un nuevo bloque llamado "Vista completa"... y arrástralo a cualquier parte de la página, ¡vaya! Ya está.

¿Qué es esta "Vista completa"? No es nada especial, de hecho, ¡es un poco redundante! No es más que un "bloque Twig" que renderiza el bloque llamado `body`. Así que, sí, podríamos haber hecho esto fácilmente utilizando el bloque Twig normal y escribiendo `body`.

Publica este diseño... y luego ve a "Mapeos de diseño". Añade una nueva... y esta vez la enlazaré primero... a "Diseño de la lista de recetas". Luego haz clic en "Detalles". Como la última vez, podríamos mapear esto a través del nombre de la ruta. Pero para ver algo diferente, utiliza "Información de la ruta", que, de nuevo, es sólo una palabra elegante para la URL, pero sin ningún parámetro de consulta. Haz que coincida con `/recipes`... "Guarda los cambios" y... ¡bien!

Cuando probamos la página... ¡se ve genial! Excepto que, ¡me olvidé del nav y del pie de página! Añadir esos dos bloques al "Diseño de la lista de recetas" es fácil. Pero ¿qué pasa si, más adelante, decidimos que cada página debe mostrar tanto el bloque de navegación en la parte superior como un banner dinámico, quizá para una venta que estemos realizando? Si eso ocurriera, tendríamos que editar cada diseño para añadir manualmente ese nuevo banner.

Diseños compartidos

Afortunadamente, hay una forma mejor de manejar elementos de diseño repetidos como éste.

Pulsa "Descartar" para volver a la lista de diseños, y luego haz clic en "Diseños compartidos" y "Nuevo diseño compartido". Como siempre, el tipo de diseño no importa mucho, así que usaré el normal... y lo llamaré "Diseño de navegación y pie de página".

Este no va a ser un diseño real que esté vinculado a ninguna página. No, sólo va a ser una maqueta de la que robaremos piezas. En la zona superior, crea un bloque de Twig que se muestre en `navigation`... e incluso lo etiquetaré como "Top Nav" para que quede más claro. A continuación, en cualquier otra zona -puedes ponerla en la parte inferior, pero no es necesario-, añade otro bloque Twig que renderice `footer` y se etiquete como Pie de página.

¡Genial! Pulsa "Publicar diseño". Ahora tenemos un diseño compartido. De nuevo, no están pensados para ser asignados a páginas: están pensados para que los utilicemos en otros diseños reales.

Compruébalo: edita el "Diseño de la lista de recetas". En la parte inferior izquierda de la pantalla, escondido detrás de la barra de herramientas de depuración web -la cerraré temporalmente- hay un botón para vincular una zona con una zona de diseño compartido. Haz clic en él y selecciona la zona superior... llamada zona "Cabecera", aunque ese nombre no es importante.

Ahora, podemos encontrar una zona compartida desde un diseño compartido... y sólo tenemos una. Pulsa "Seleccionar zona" y... ¡ya está! La zona superior de nuestro diseño será ahora igual al bloque o bloques que haya en la zona superior de ese diseño compartido. Si añadimos más cosas a esa zona en el diseño compartido, aparecerán automáticamente aquí.

Hazlo una vez más: selecciona la última zona para que el pie de página aparezca definitivamente en la parte inferior, selecciona la zona compartida y... ¡listo!

Publica eso, muévete, actualiza y... ¡la página completa está de vuelta! Repitamos rápidamente esto para la "Disposición de la página de inicio". Pero esto es complicado porque pongo todos mis bloques dentro de esta zona superior. En general, estas zonas no importan, pero en este caso, para evitar sobrescribir todo esto, arrastraré todo excepto el bloque Twig de navegación hacia abajo. Podemos arreglar el orden más tarde.

Y ahora, configura la zona superior para que utilice la del diseño compartido. Sí Reemplaza lo que teníamos allí antes. A continuación, enlaza también la zona inferior con el diseño compartido.

¡Perfecto! Comprobemos el orden de nuestros bloques... que es lo bueno de los diseños. Si no me gusta el orden de lo que hay en mi página, ¡siempre puedo cambiarlo! Eso es mejor. Publica el diseño, vuelve a la página de inicio en el frontend y... ¡bien!

Siguiente: hagamos que nuestra página de la lista de recetas sea más flexible permitiendo que esta zona superior de `h1` se construya y personalice desde dentro de los diseños... en lugar de que esté codificada en la plantilla.

Chapter 5: Añadir más bloques personalizados

Vamos a trabajar más en este Diseño de Lista de Recetas más adelante. Pero, vamos a hacer una cosa más ahora. Editar este diseño. Quiero dar a nuestros usuarios administradores la flexibilidad de cambiar este título. ¡Genial! Añadamos un nuevo bloque de título justo encima... e introduzcamos algo de texto.

Pulsa "Publicar y continuar editando"... y luego ve al frontend. Lo que intento hacer es replicar este título, o área "héroe", para poder eliminarlo de nuestra plantilla Twig. Pero cuando refrescamos, todavía no se ve bien.

Ve y mira esa plantilla. Bien: para replicar esto, necesitamos una etiqueta `h1` envuelta en un div `hero-wrapper`:

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 %}
```

Ahora mismo, Layouts está simplemente renderizando un `h1`. Y, por cierto, puedes, en las opciones del bloque de título, elegir entre `h1`, `h2`, o `h3`. `h1` es lo que necesitamos esta vez.

Añadir una columna Div envolvente

Entonces: ¿cómo podemos envolver esto en un `div` y darle una clase `hero-wrapper`? La respuesta: añadiendo un ingenioso bloque "columna"... y luego moviendo el título a esa columna. Genial, ¿verdad? Por último, al hacer clic en la columna, puedes añadir la clase que quieras. Añade `hero-wrapper`.

¡Vamos a probarlo! Pulsa "Publicar y continuar editando", actualiza el frontend y... ¡mucho mejor! ¿Qué pasa con ese texto? Cópialo, añade un nuevo bloque de "texto" justo debajo de

nuestro "título" y... pega. Publica y continúa editando de nuevo... prueba de nuevo el frontend y... ¡mira eso! ¡Una réplica perfecta!

Para celebrarlo, en la plantilla, podemos eliminar esa sección por completo:

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
```

El resultado final es el mismo que antes... excepto que los usuarios administradores tienen ahora la posibilidad de cambiar el texto.

¿CSS personalizado en las plantillas o tipo de bloque personalizado prefabricado?

Sin embargo, probablemente te hayas dado cuenta de que esto me obligó a ser un poco técnico: tuve que conocer la clase CSS que necesitaba la columna. Si los usuarios administradores que diseñan tus diseños son un poco técnicos, entonces esto podría no ser un problema. Pero si tus editores son menos técnicos, podrías, en cambio, crear un tipo de bloque personalizado -como un bloque de héroe- en el que el usuario sólo tenga que escribir el texto y tú lo renderices todo por él. No vamos a crear bloques personalizados en este tutorial... pero eso es sobre todo porque, al final del tutorial, sabrás todo lo que necesitas para seguir los documentos para ello.

La barra de herramientas de depuración de Layouts Web

Muy bien, de vuelta en el front-end, Layouts viene con su propio icono de la barra de herramientas de depuración web. Y si haces clic en esto, es bastante genial. Vamos a usar esto un montón de veces. Te muestra el diseño que se ha resuelto e incluso la razón por la que se ha elegido.

Pero lo realmente útil es la sección "Bloques renderizados". Esto nos muestra todos los bloques de diseño que se renderizaron para construir esta página. Puedes ver que hay uno llamado "Bloque Twig" para la navegación superior, una "Columna", luego el bloque "Título",

"Texto", "Vista completa" y finalmente el último bloque "Twig" para el pie de página. Esta es una gran manera de ver todos los diferentes bloques que se están renderizando, así como la plantilla que hay detrás de cada uno. Más adelante, hablaremos de cómo anular esas plantillas, para poder personalizar su aspecto.

Vinculación con el administrador de diseños

De vuelta al administrador de Diseños, publica el diseño para volver a la página principal. Si vas a `/admin`, verás que nuestra aplicación ya tiene EasyAdmin instalado. Vamos a añadir un enlace desde el menú de aquí a Diseños para hacer la vida más fácil.
Abre `src/Controller/Admin/DashboardController.php`... y encuentra `configureMenuItems()`. Añade otro con `yield MenuItem::linkToUrl()`, llámalo "Layouts" y dale unos iconos: `fas fa-list`. Para la url, utiliza `this->generateUrl()` y pasa el nombre de la ruta, que resulta ser `nglayouts_admin_layouts_index`:

src/Controller/Admin/DashboardController.php

```
↕ // ... lines 1 - 12
13 class DashboardController extends AbstractDashboardController
14 {
↕ // ... lines 15 - 34
35     public function configureMenuItems(): iterable
36     {
↕ // ... lines 37 - 38
39         yield MenuItem::linkToUrl('Layouts', 'fas fa-list', $this->generateUrl('nglayouts_admin_layouts_index'));
40     }
41 }
```

¡Perfecto! Es un pequeño detalle, pero ahora, cuando estemos en `/admin`, podemos hacer clic en "Diseños" para saltar directamente allí.

Bien, ¡comprobación de estado! Podemos representar los bloques Twig y mezclar títulos, texto, HTML, Google Maps y otros bloques donde queramos. Cuantos más bloques Twig tengamos en la plantilla, más flexibilidad tendremos.

¿Pero qué pasa con la posibilidad de representar una colección de recetas de nuestra base de datos, como las "Últimas recetas" que tenemos en la página de inicio? Esa es una pieza importante de los diseños: así que empecemos a sumergirnos en ella a continuación.

Chapter 6: Añadir listas: Tipo de valor

Tenemos una entidad `Recipe` y, en el frontend, una página que enumera las recetas. También hemos visto lo fácil que es crear un diseño, que hace que partes de esta página sean configurables al instante.

¿Añadir listas de contenido existente a través de maquetas?

Pero ahora, viendo la página de inicio, me pregunto si podemos añadir bloques más complejos, más allá del simple texto. ¿Podríamos, por ejemplo, añadir un bloque que muestre una lista de recetas? ¿Algo similar a lo que tenemos aquí ahora... excepto que en lugar de añadirlo a través de un bloque Twig, se añade completamente a través de diseños por un usuario administrador? Y, para ir más lejos, ¿podríamos incluso dejar que el usuario administrador eligiera qué recetas mostrar aquí?

¡Totalmente! Si la primera gran idea de Layouts es permitir que los bloques de plantillas Twig se reorganicen y se mezclen con contenido dinámico, entonces la segunda gran idea es permitir que los usuarios administradores incrusten en nuestra página piezas de contenido existente, como las recetas de nuestra base de datos.

¿Cómo? Edita el diseño de la página de inicio. En los bloques de la izquierda, fíjate en este llamado "Rejilla". Añádelo después de nuestro bloque Twig "Héroe". La cuadrícula nos permite añadir elementos individuales a ella... que podrían ser cualquier cosa. Pero, ¡no veo la forma de hacerlo!

Vale, sabemos que muchos bloques, como los títulos, los mapas, el markdown, etc., pueden añadirse a nuestras páginas en los diseños de forma inmediata, sin ningún trabajo de configuración adicional. Pero el propósito de algunos bloques -como el de Lista, el de Cuadrícula y el de Galería aquí abajo (que no son más que cuadrículas extravagantes que tienen un comportamiento de JavaScript asociado a ellas)- es representar una colección de "elementos" que se cargan desde otro lugar, como nuestra base de datos local, el CMS o incluso tu tienda Sylius. Las "cosas" o "elementos" que podemos añadir a estos bloques se llaman "tipos de valores". Y... actualmente tenemos cero. Si se tratara de un proyecto de Sylius,

podríamos instalar la integración de Sylius y Layouts y al instante podríamos seleccionar productos. Lo mismo ocurre si utilizas Ibexa CMS.

Añadir un tipo de valor

Éste es nuestro siguiente gran objetivo: añadir nuestra entidad Doctrine `Recipe` como "tipo de valor" en los diseños para poder crear listas y cuadrículas que contengan recetas.

El primer paso para añadir un tipo de valor es informar a Layouts sobre él en un archivo de configuración. En `config/packages/netgen_layouts.yaml`, de forma muy sencilla, di `value_types`, y debajo, `doctrine_recipe` . Este es el nombre interno del tipo de valor, y nos referiremos a él en algunos lugares. Dale un nombre amigable para los humanos `name` - `Recipe` - y por ahora, pon `manual_items` a `false`... y asegúrate de que tiene una "s" al final:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 3
    value_types:
        doctrine_recipe:
            name: Recipe
            manual_items: false
```

Hablaremos más tarde de `manual_items`, pero es más fácil ponerlo en `false`para empezar.

Dirígete, actualiza nuestra página de diseños (no pasa nada por recargarla)... ¡y echa un vistazo a nuestro bloque Grid! Hay un nuevo campo "Tipo de colección" y "Colección manual" es nuestra única opción ahora mismo. Entonces... parece que esto sigue sin funcionar. No puedo cambiar esto por otra cosa... y tampoco puedo seleccionar elementos manualmente.

Consultas dinámicas vs manuales

Hay dos formas de añadir elementos a uno de estos bloques de "colección". La primera es una colección dinámica en la que elegimos a partir de una consulta preelaborada. Podríamos elegir una consulta "Más populares" que buscara las recetas más populares o una consulta "últimas recetas", por poner dos ejemplos. La segunda forma de elegir elementos es manual: el usuario administrador selecciona literalmente los que quiere de una lista.

Añadir un tipo de consulta

Vamos a empezar con el primer tipo: la colección dinámica. Todavía no vemos la opción "Colección dinámica" porque primero tenemos que crear una de esas consultas prefabricadas. Esas consultas prefabricadas se llaman `query_types`. Podríamos, por ejemplo, crear un tipo de consulta para `Recipe` llamado "Más popular" y otro llamado "Más reciente".

¿Cómo las creamos? Vuelve al archivo de configuración, añade `query_types` y debajo, digamos, `latest_recipes`. Una vez más, esto es sólo un "nombre interno". También dale un nombre legible para los humanos `name`: `Latest Recipes`:

config/packages/netgen_layouts.yaml

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

Entonces... ¿qué hacemos ahora? Si volvemos atrás y refrescamos... obtenemos un error muy bonito que nos dice qué hacer a continuación:

> *"El gestor de tipos de consulta para el tipo de consulta `latest_recipes` no existe."*

¡Está intentando decirnos que tenemos que construir una clase que represente este tipo de consulta! ¡Hagámoslo!

La clase manejadora del tipo de consulta

En el directorio `src/`, voy a crear un nuevo directorio `Layouts/`: aquí organizaremos muchas de nuestras cosas de Layouts personalizados. A continuación, añade una nueva clase PHP llamada... qué tal `LatestRecipeQueryTypeHandler`. Haz que esto implemente `QueryTypeHandlerInterface`:

src/Layouts/LatestRecipeQueryTypeHandler.php

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

Luego ve a "Generar código" (o `Command`+`N` en un Mac), y selecciona "Implementar métodos" para añadir los cuatro que necesitamos:

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.
    }
}
```

¡Bien! Veamos... Dejaré `buildParameters()` vacío por un momento, pero volveremos a él pronto:

src/Layouts/LatestRecipeQueryTypeHandler.php

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

El método más importante es `getValues()`. Aquí es donde cargaremos y devolveremos los "artículos". Si nuestras recetas estuvieran almacenadas en una API, haríamos aquí una petición a la API para obtenerlas. Pero como están en nuestra base de datos local, las consultaremos.

Para ello, ve a la parte superior de la clase, añade un método `__construct()` con `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
}
```

A continuación, baja a `getValues()`, `return $this->recipeRepository`... y utiliza un método que ya he creado dentro de `RecipeRepository` llamado `->createQueryBuilderOrderedByNewest()`. Añade también `->setFirstResult($offset)` y `->setMaxResults($limit)`. El usuario administrador podrá elegir cuántos elementos mostrar e incluso podrá saltarse algunos. Y así, Layouts nos pasa esos valores como `$limit` y `$offset`... y los utilizamos en nuestra consulta. Terminamos con `->getQuery()` y `->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
}
```

¡Perfecto! A continuación, para `getCount()`, vamos a hacer exactamente lo mismo... excepto que no necesitamos `->setMaxResults()` ni `->setFirstResult()`. En su lugar, añadimos `->select('COUNT(recipe.id)')`:

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()
// ... line 34
    }
// ... lines 36 - 40
}
```

Estoy utilizando `recipe` porque, en `RecipeRepository`... si miramos el método personalizado, utiliza `recipe` como alias en la consulta:

src/Repository/RecipeRepository.php

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

Después, actualiza `->getResult()` para que sea `->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
}
```

¡Uf! Ha sido un poco de trabajo, pero bastante sencillo. Ah, y para `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;
    }
}
```

No lo vamos a necesitar, pero este método es bastante chulo. Si devuelve `true`, puedes leer la información de la página actual para cambiar la consulta, como si estuvieras en una página de "categoría" y necesitaras listar sólo los productos de esa categoría.

Etiquetar la clase manejadora del tipo de consulta

De todos modos, eso es todo. ¡Esto es ahora un manejador de tipos de consulta funcional! Pero si vuelves a actualizarlo... sigue sin funcionar. Nos da el mismo error. Esto se debe a que tenemos que asociar esta clase de manejador de tipos de consulta con el tipo de consulta `latest_recipes` en nuestra configuración. Para ello, tenemos que dar una etiqueta al servicio... y hay una forma muy interesante de hacerlo gracias a Symfony 6.1.

Sobre la clase, añade un atributo llamado `#[AutoconfigureTag()]`. El nombre de la etiqueta que necesitamos es `netgen_layouts.query_type_handler`: está sacado de la documentación. También necesitamos pasar un array con una clave `type` establecida en `latest_recipes`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
// ... lines 1 - 8
use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;

#[AutoconfigureTag('netgen_layouts.query_type_handler', ['type' =>
'latest_recipes'])]
class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
{
// ... lines 14 - 42
}
```

Este `type` debe coincidir con lo que tenemos en nuestra configuración:

config/packages/netgen_layouts.yaml

```
netgen_layouts:
// ... lines 2 - 8
    query_types:
        latest_recipes:
// ... lines 11 - 12
```

Esto une a los dos.

Y ahora... ¡la página funciona! Si hacemos clic en nuestro bloque Grid... podemos cambiar a "Colección dinámica". ¡Espectacular! Le doy a Aplicar y... ¡todo deja de cargarse

inmediatamente!

Cuando tengas un error en la sección de administración, es muy probable que aparezca a través de una llamada AJAX. A menudo, los diseños te mostrarán el error en un modal. Pero si no lo hace, no te preocupes: sólo tienes que mirar aquí abajo en la barra de herramientas de depuración web. ¡Sí! Tenemos un error 400.

Vamos a solucionarlo creando un convertidor de valores. Luego haremos nuestra consulta aún más inteligente.

Chapter 7: Convertidor de valores

En cuanto cambiamos nuestro tipo de cuadrícula para utilizar una colección dinámica... dejó de cargarse. El error se esconde aquí abajo en esta llamada AJAX. La mejor manera de verlo es abrir esa URL en una nueva pestaña. Ahí lo tenemos:

> *"El convertidor de valores para el tipo `App\Entity\Recipe` no existe."*

Bien, hasta ahora hemos creado un "tipo de valor" personalizado para `Recipe`, que no era más que esta configuración, y un "tipo de consulta" personalizado que nos permite cargar una lista de las últimas recetas ejecutando la consulta dentro de la clase asociada. Ahora recibimos este error del convertidor de valores.

Creación de la clase convertidor de valores

Un convertidor de valores es realmente sencillo: es una clase que transforma el objeto subyacente - `Recipe` - en un formato que los Layouts puedan entender. En ese mismo directorio `src/Layouts/`, vamos a crear una clase `RecipeValueConverter`... y hacer que implemente `ValueConverterInterface`:

src/Layouts/RecipeValueConverter.php

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

use Netgen\Layouts\Item\ValueConverterInterface;

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

Ya conoces el procedimiento: ve a "Código" -> "Generar" (o `Command`+`N` en un Mac) y pulsa "Implementar métodos" para generar los siete que necesitamos:

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.
    }
}
```

Lo sé, parece mucho, pero son súper fáciles de rellenar.

En primer lugar, para `supports()`, Layouts llamará a este método cada vez que tenga un "valor" que no entienda. Queremos decirle que sabemos cómo convertir el `$object` si es un `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
}
```

En segundo lugar, para `getValueType()`, `return` la clave interna de nuestro tipo de valor: `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
}
```

Lo siguiente es `getId()`... y literalmente vamos a `return` nuestro ID con `$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
}
```

No tenemos autocompletado en esto, pero sabemos que este objeto será un `Recipe`.

Para `getRemoteId()`, sólo `return $this->getId($object)`:

src/Layouts/RecipeValueConverter.php

```
// ... lines 1 - 7
class RecipeValueConverter implements ValueConverterInterface
{
// ... lines 10 - 24
    public function getRemoteId(object $object)
    {
        return $this->getId($object);
    }
// ... lines 29 - 45
}
```

Este método sólo es importante si planeas utilizar la función de importación en Layouts para mover los datos, por ejemplo, entre la puesta en escena y la producción. Si piensas hacerlo, puedes dar a tus objetos un UUID y devolverlo aquí.

Aquí abajo, para `getName()`, éste será un nombre legible para los humanos que se mostrará en el área de administración. Esta vez, para ayudar a mi editor, vamos a `assert()` que `$object instanceof Recipe`:

src/Layouts/RecipeValueConverter.php

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

Dos cosas sobre esto. Primero, sabemos que este objeto será siempre un `Recipe` porque, arriba en `supports()`, dijimos que es el único objeto que soportamos. En segundo lugar, si no has visto la función `assert()` antes, si el `$object` no es un `instanceof` `Recipe` , lanzará una excepción. Es una forma muy fácil de decirle a tu editor -u otras herramientas como PHPStan- que el objeto es definitivamente una instancia de `Recipe`.... , lo que significa que ahora obtendremos el autocompletado cuando digamos `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 }
```

Lo siguiente es `getIsVisible()`. Sólo `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 }
```

Si tus recetas pudieran ser publicadas o no, por ejemplo, entonces podrías comprobarlo aquí para devolver `true` o `false`.

Por último, para `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 }
```

Lo sé, parece una tontería, pero es una forma práctica de cambiar tu `Recipe` después de que se haya cargado, si lo necesitas. Pero eso no es algo que necesitemos hacer.

Y... ¡listo!

Esta vez, a diferencia del manejador de tipos de consulta, la autoconfiguración se encarga de todo... así que no necesitamos añadir una etiqueta manual aquí arriba. Observa: muévete y prueba a refrescar la ruta AJAX primero. ¡Eso funciona! Ahora vete, refresca la página de administración de los diseños... y guau. ¡Fíjate! ¡Vemos un montón de elementos en nuestra parrilla! Si hacemos clic en él, vemos que los elementos se cargan abajo. ¡Es increíble!

Personalizar los resultados de los artículos

Fíjate en que sólo hemos tenido que elegir "colección dinámica". Nosotros... nunca le dijimos al sistema que queríamos utilizar el tipo de consulta "últimas recetas". Esto se debe simplemente a que sólo tenemos un tipo de consulta... y por lo tanto, Layouts adivinó que eso era lo que queríamos. Si añadiéramos un segundo tipo de consulta al sistema, veríamos aquí otro desplegable de selección en el que podríamos elegir entre las últimas recetas y las recetas "más populares", por ejemplo.

Así que estamos utilizando nuestro tipo de consulta "últimas recetas" para obtener 25 resultados. Si intentáramos recrear esta zona aquí, sólo querríamos 4. Así que vamos a limitar el número de elementos a cuatro. ¡Genial!

Comprobando el Frontend

¿Qué aspecto tiene esto en el frontend? ¡Averigüémoslo! Pulsa "publicar y continuar editando" y.... una vez que se guarde, ve y actualiza. Debería aparecer aquí, pero... ¡no vemos absolutamente nada! O... eso parece al principio.

Pero cuando inspeccionamos el elemento... y hacemos un poco de zoom... hay un `div` con la clase `ngl-vt-grid` en él. Y dentro, una fila y dentro de ésta, un montón de divs vacíos. Si ignoras los elementos de `clearfix`, ¡esto renderiza 1, 2, 3, 4 divs para nuestros cuatro elementos! Así que los elementos se renderizan... simplemente se renderizan vacíos.

Y eso tiene sentido. Todavía no le hemos dicho a los layouts cómo deben representarse los elementos de la receta. En unos minutos hablaremos de ello.

Opciones del formulario del tipo de consulta (parámetros)

Pero antes de llegar allí, quiero hacer que nuestro tipo de consulta sea un poco más elegante. En la primera pasada, ignoramos el método `buildParameters()`. Resulta que es una forma de añadir campos de formulario adicionales para que un usuario administrador pueda pasar opciones a la consulta.

Por ejemplo, vamos a añadir un término de búsqueda opcional. Digamos que `$builder->add()` pasa a `term` - que será el nombre interno de este nuevo parámetro - y luego `TextType`: el de `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 }
```

Hay un montón de otros tipos de campos para URLs, fechas y más.

Sólo con esto, cuando refresquemos la sección de administración... y hagamos clic abajo en la cuadrícula, ¡ahí está! ¡Tenemos un gran cuadro nuevo! Por supuesto, si escribimos algo dentro, no pasa nada... y además tiene una etiqueta rara.

Traducir la etiqueta del campo

Primero vamos a arreglar esa etiqueta. Layouts tiene por defecto esta extraña cadena, pero ya la está pasando por el traductor a través de un dominio llamado `nglayouts`. Así que, en el directorio `translations/`, crea un archivo llamado `nglayouts.en.yaml`, o utiliza el formato que quieras.

Pega la etiqueta y ponla como "Término de búsqueda":

translations/nglayouts.en.yaml

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

Prueba ahora la sección de administración. Cuando hagamos clic... ¡mucho mejor! Si sigues viendo la etiqueta antigua, prueba a borrar la caché:

```
symfony console cache:clear
```

A veces Symfony no se da cuenta cuando añades un nuevo archivo de traducción.

Utilizar el parámetro

Bien, para utilizar el término de búsqueda, dirígete a nuestro manejador de tipo de consulta. El objeto `Query` que se pasa a `getValues()` contiene los parámetros que hemos añadido. Además, ¡ya he preparado el método `createQueryBuilderOrderedByNewest()` para que acepte un término de búsqueda opcional! Pasa este `$query->getParameter()`, su nombre - `term` - y luego `->getValue()`:

src/Layouts/LatestRecipeQueryTypeHandler.php

```
↕ // ... lines 1 - 12
13 class LatestRecipeQueryTypeHandler implements QueryTypeHandlerInterface
14 {
↕ // ... lines 15 - 23
24     public function getValues(Query $query, int $offset = 0, ?int $limit = null): iterable
25     {
26         return $this->recipeRepository->createQueryBuilderOrderedByNewest($query->getParameter('term')->getValue())
↕ // ... lines 27 - 30
31     }
↕ // ... lines 32 - 44
45 }
```

Copia eso y repítelo aquí abajo para el método `getCount()`:

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
}
```

Muy bien, ¡vamos a probar esto! Refresca la zona de Diseños, baja aquí y ¡creo que ha funcionado! No muestra ningún elemento... porque he utilizado un término de búsqueda bastante tonto. Despeja la zona. Lo tenemos todo. Ahora escribe unas cuantas letras... y observa cómo cambia abajo.

A continuación, vamos a enseñar a los diseñadores cómo mostrar los elementos de la receta tanto en el frontend como en la vista previa del área de administración.

Chapter 8: Plantilla de la vista del artículo

Bien, equipo, las cosas tienen buena pinta. Hemos creado un "tipo de valor" de `Recipe`, una consulta personalizada para cargarlos y un convertidor de valores para ayudar a los layouts a entender nuestros objetos de `Recipe`.

Lo que aún no hemos hecho es decirle a Layouts cómo representar un elemento `Recipe`, siendo elemento la palabra que Layouts utiliza para las "cosas" individuales que los bloques de rejilla y lista representan. Y, de hecho, tenemos que decirle a Layouts cómo representar una versión de administrador de un elemento de receta, que se mostrará aquí, así como la versión más importante del elemento en el frontend.

Añadir una vista de elemento

La forma en que se muestra un elemento se denomina "vista de elemento". Para añadir una nueva vista de artículo, empezaremos en la configuración. Añade una clave `view` con `item_view` debajo y app debajo. Añadiré un comentario, porque en Layouts, `app` significa "admin". Así que lo que vamos a definir bajo la clave `app` será la vista de administrador para nuestro elemento de receta:

config/packages/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 12
13     view:
14         item_view:
15             # app = admin
16             app:
↕  // ... lines 17 - 22
```

A continuación, añade `recipes_app`... con una pequeña nota para decir que esta clave no es importante:

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
```

A diferencia de otras cosas, como `latest_recipes`, esta clave interna no se utilizará en ningún sitio. A continuación, necesitamos dos cosas importantes. En primer lugar, `template` - no incluyas la "s" como he hecho yo- pon una ruta de acceso a la plantilla, como `nglayouts/` - ese es un nombre de directorio estándar para usar en las plantillas, pero podrías usar cualquier cosa-, y luego, ¿qué tal `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
```

La segunda cosa importante es la clave muy especial `match`. Tenemos que decirle a Layouts que ésta es la plantilla que debe utilizarse cuando se está renderizando un elemento de la receta. Por ejemplo, imagina que tenemos dos tipos de valores: recetas y también entradas de blog. Pues bien, Layouts necesitaría saber que ésta es la plantilla que debe usarse para las recetas... pero que debe usar una plantilla de elemento diferente para las entradas del blog.

La clave de configuración "match"

Para ello, utilizaremos una sintaxis extraña: `item\value_type` ajustado a `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'
```

Donde `doctrine_recipe` hace referencia al nombre de nuestro tipo de valor aquí arriba:

config/packages/netgen_layouts.yaml

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

Vamos a ver esta clave `match` varias veces más en este tutorial. Layouts tiene un montón de "emparejadores" incorporados, que se identifican con cadenas como `item\value_type`. Se utilizan para ayudar a emparejar una pieza de configuración, como esta plantilla, con otra pieza de configuración, como el tipo de valor `doctrine_recipe`. Hay un número finito de estos emparejadores, y vamos a ver los más importantes a lo largo del camino. Así que no te preocupes demasiado por ellos.

Ah, pero déjame que corrija mi error tipográfico: debería ser `template` sin "s".

Los dos tipos de vista: item_view y block_view

De todos modos, quiero mencionar una cosa rápida sobre la clave de configuración `view`: sólo hay un pequeño número de subclaves que van debajo de ella.

Busca tu terminal y ejecuta:

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

Esto volcará una enorme lista de config, pero no te abrumes Veremos las partes importantes de esto más adelante. Lo que quiero que mires son las claves raíz que van por debajo de `view`, como `block_view` y `layout_view`.

Resulta que hay seis claves diferentes que puedes poner debajo de la clave `view` en tu configuración, pero sólo nos interesan dos de ellas... y por eso lo menciono. Cuando se trata de personalizar tus vistas, ¡es realmente muy sencillo! La primera clave de la que nos ocupamos es `item_view`, que controla las plantillas que se utilizan cuando se representan "elementos": es decir, cuando se representan cosas dentro de una cuadrícula o una lista. La otra subclave de la que nos ocupamos es `block_view`, que es la forma de configurar la plantilla que se utiliza para representar diferentes tipos de bloques, como el bloque `title` o el bloque `text`.

Sí, o bien estás renderizando un bloque y quieres personalizar su plantilla o bien estás renderizando un elemento dentro de un bloque y quieres personalizar la plantilla de ese elemento. Así que la configuración parece gigantesca, pero la mayoría de estas cosas son internas y nunca tendrás que preocuparte por ellas.

Creación de la plantilla de administración

Bien: tenemos nuestro `item_view` para nuestro `doctrine_recipe` para el área de administración. Vamos a añadir esa plantilla. En el directorio `templates/`, crea dos nuevos subdirectorios: `nglayouts/admin/`. Y luego, un nuevo archivo llamado `recipe_item.html.twig`. Dentro, escribe `Does it work?` y... usemos también la función `dump()` para poder ver a qué variables tenemos acceso:

templates/nglayouts/admin/recipe_item.html.twig

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

Bien, vuelve a tu navegador, actualiza el administrador de diseños y... ¡funciona! Y, aparentemente, tenemos acceso a varias variables. La más importante es `item`. ¡Se trata de un objeto `CmsItem` de Layouts... y tiene una propiedad llamada `object` establecida a nuestro `Recipe`!

¡Vamos a utilizarlo! Digamos `{{ item.object.name }}`, luego una tubería, y... imprimamos también una fecha: `{{ item.object.createdAt }}` - una de las otras propiedades de `Recipe` canalizada en el filtro `date` con `Y-m-d`:

templates/nglayouts/admin/recipe_item.html.twig

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

¡Vamos a comprobarlo! Muévete, refresca y... ¡ya está! Puedes hacer esto más elegante si quieres, pero esto nos servirá.

A continuación: vamos a crear la vista de artículos del frontend.

Chapter 9: Vista de artículos del frontend

Es hora de crear la vista de artículos `Recipe` para el frontend. Esto empieza casi exactamente igual. De hecho, copia la configuración del admin... y pégala. En Layouts, sabemos que la clave `app` significa la sección "admin". Y resulta que `default` se utiliza para significar el 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 (por defecto) item_view & Template

Una vez más, este nombre interno no es importante, para la plantilla, utiliza la misma ruta pero `frontend`... y mantén `match` exactamente igual:

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'
```

¡Me encanta cuando las cosas son aburridas y fáciles! Vamos a crear esa plantilla. En `nglayouts/`, haz el directorio `frontend/`... y dentro, `recipe_item.html.twig`.

A esta plantilla le pasaremos las mismas variables que a la plantilla de elementos de administración. Esto significa que podemos, una vez más, utilizar `{{ item.object }}` para acceder a nuestro objeto `Recipe`. Vamos a imprimir la clave `name` para ver si las cosas funcionan:

templates/nglayouts/frontend/recipe_item.html.twig

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

Y... están funcionando. ¡Está vivo!

Comprobando las plantillas en el perfil de Twig

Una de mis cosas favoritas cuando empiezo a trabajar con plantillas dentro de Layouts es hacer clic en el elemento Twig de la barra de herramientas de depuración web. Aquí podemos ver realmente cómo está renderizando Layouts. Sí, renderiza `layout_2.html.twig`... y luego empieza a renderizar cada zona. Renderiza nuestro bloque `navigation`, el bloque `hero`, y luego, finalmente aquí abajo, la cuadrícula. Puedes ver que está usando `grid/3_columns.html.twig`. Esto es algo que podemos controlar en el área de administración. Haz clic en la cuadrícula. A la derecha, vemos la pestaña "Contenido". Pero también hay una pestaña "Diseño". Cambia esto a "4 columnas"... y le doy a "Publicar y continuar editando".

Si ahora refrescamos y volvemos a cargar el perfilador Twig, veremos que se renderiza `4_columns.html.twig`. Entonces, ¡eh! Dentro de cada columna, renderiza nuestro `recipe_item.html.twig`. Esto es realmente genial de ver, y vamos a ver esto de nuevo más tarde cuando hablemos de anular las plantillas del núcleo.

CSS de Bootstrap 4

Una cosa que tengo que mencionar es que nuestra aplicación está utilizando la versión 4 de Bootstrap, no Bootstrap 5. La razón es que, en este momento, la plantilla de cuadrícula muestra el marcado de la versión 4 de Bootstrap. Si quisieras usar Bootstrap 5, es totalmente posible, pero tendrías que anular estas plantillas de columnas -como `4_columns.html.twig` - para modificar las clases. Anular las plantillas del núcleo es en realidad súper fácil, y hablaremos de cómo hacerlo pronto.

Personalizar nuestra plantilla frontal

Bien, ¡vamos a dar vida a esta vista frontal! Abre la plantilla de la página de inicio: `main/homepage.html.twig`... y desplázate hacia arriba hasta el lugar en el que hacemos un bucle con las últimas recetas. Perfecto. Lo que básicamente quiero hacer es robar el marcado de uno de estos mosaicos de recetas... y pegarlo en la plantilla del frontend:

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>
```

Ahora sólo tenemos que ajustar algunas variables: en lugar de `recipe.slug`, tiene que ser `item.object.slug`. Haré una búsqueda y sustitución: sustituye `recipe.` por `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>
```

Envolver los bloques en un contenedor

¡Muy bien! Veamos si ha funcionado. Muévete, refresca... ¡y lo hizo! Esto parece el frontend. ¡Somos increíbles! Excepto que falta el "canalón" que tenemos en el original. Inspecciona el elemento. Ah, la diferencia es que las columnas originales estaban dentro de un div `container`, que añade el margen. En el nuevo código, estamos dentro de una fila... pero no de un `container`.

Para arreglar esto en Layouts, vamos a añadir nuestro bloque de utilidad favorito: ¡una columna! Mueve la cuadrícula dentro de esa columna. Entonces, podríamos añadir una clase CSS como hicimos antes en la zona del héroe. Pero en su lugar, toma un atajo y marca "Envolver en contenedor".

Pulsa "Publicar y continuar editando" y actualiza. Vaya, página equivocada. Vuelve a la página de inicio y... ¡se ve muy bien! ¡Ahora está dentro de un elemento con una clase `container`!

Esta "envoltura en el contenedor" es súper útil: añade literalmente un `div` extra alrededor de tu bloque con `class="container"` y todos los bloques lo admiten. Diablos, ni siquiera necesitamos una columna: podríamos haber marcado simplemente la opción "Envolver en contenedor" en la propia cuadrícula.

La única razón por la que he puesto esto dentro de una columna es para que también podamos añadir allí la cabecera "Últimas recetas". Arrastra un nuevo bloque "Título" dentro de la columna. ¡Sal de aquí, Apple! Dentro, escribe "Últimas recetas" y cambia a un `h2`.

Pulsa nuestro favorito "Publicar y continuar editando", actualiza y... ¡aún más cerca! Sólo tenemos que centrar esto... y quizás darle un pequeño margen superior. Añade dos clases al título: `text-center` y `my-5` para darle un poco de margen vertical: ambas clases provienen de Bootstrap. Sólo estoy repitiendo las clases que mi diseñador ya utilizaba en la plantilla.

Publica eso... y cuando lo probamos... coincide exactamente. ¡Guau! Pero ahora, ¡tenemos un control total sobre las recetas que hay dentro! Podemos cambiar a una consulta diferente, cambiar el número de elementos o, dentro de un rato, podemos optar por seleccionar manualmente las recetas exactas a mostrar. Ahora también podemos incrustar listas y cuadrículas de recetas en cualquier lugar que queramos del sitio.

¡Limpieza!

Para celebrarlo, elimina todo el bloque Twig de `latest_recipes`:

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
```

Y, arriba en `MainController`, elimina la consulta, la variable, el argumento del repositorio y la declaración `use`:

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', [
        ]);
    }
}
```

Cuando actualicemos, sólo tendremos una sección "Últimas recetas" procedente de nuestro bloque dinámico. Ah, pero fíjate en que en el admin de layouts, seguimos mostrando el bloque `latest_recipes`... ¡aunque ya no exista! Layouts es bastante indulgente con los usuarios administradores: en lugar de lanzar un error, simplemente no renderiza nada.

Pero borremos eso... luego publiquemos... y echemos un último vistazo. ¡Me encanta!

A continuación: ahora que tenemos esta cuadrícula dentro de Layouts, podemos hacer algunas cosas interesantes con ella, como activar la paginación con Ajax.

Chapter 10: Paginación Ajax y CSS/JS

Ahora que estamos renderizando estos elementos de la receta a través del tipo de bloque de rejilla, comprueba lo que podemos hacer. Haz clic en la cuadrícula, ve a la pestaña de diseño y marca "Activar la paginación". Entonces podrás elegir entre un estilo de paginación con enlaces de página, como 1, 2, 3 y 4, o simplemente un botón de "cargar más". Utilicemos este último.

Muy bien, pulsa "Publicar y continuar editando". Entonces... una vez que se guarde, actualiza para ver... ¡absolutamente nada! La paginación se realiza completamente a través de JavaScript y Ajax. Y no vemos nada porque aún no hemos incluido el JavaScript necesario en nuestra página.

Incluir las plantillas CSS/JS

Añadirlo es bastante fácil. Ve a `templates/base.html.twig`. Aquí arriba, en la zona de `head`, vamos a incluir dos plantillas. La primera es:`@NetgenLayoutsStandard/page_head.html.twig`... y pasarle una variable extra: `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>
```

Esto cargará el CSS y el JavaScript que soportan estos elementos de galería aquí abajo. No voy a hablar de estos bloques de galería en este tutorial, pero son básicamente como un bloque de lista o de cuadrícula, excepto que tienen JavaScript para convertirlos en deslizadores o galerías de miniaturas.

Así que esto incluye el CSS y el JavaScript para ellos, así como un pequeño archivo CSS de cuadrícula para ayudar a representar las columnas de cuadrícula en tu página en caso de que no tengas Bootstrap. El `full: true` le dice que traiga jQuery así como otras dos bibliotecas de JavaScript llamadas `magnific-popup` y `swiper`. Todas ellas son necesarias para los bloques de la galería.

Así que, sí, si no estás usando uno de esos bloques de galería, podrías evitar incluir este archivo por completo. Pero yo lo dejaré.

Pero fíjate, no he dicho nada sobre la paginación. Para eso, necesitamos incluir una segunda plantilla. Copia esta línea, pégala, quita la palabra `Standard` y esto no necesita la variable `full`:

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>
```

Esta plantilla es muy sencilla: incorpora un poco de CSS y un poco de JavaScript para potenciar la paginación Ajax. Y éstas son las dos únicas plantillas que necesitarás incluir para las maquetas JavaScript y CSS.

Añadir la plantilla de elementos "ajax

Actualiza y... ¡ahí está! Y cuando hacemos clic en el nuevo enlace... ¡estalla con un error 500! Ups.

Abre esa URL en una nueva pestaña. Interesante:

> *“No se ha encontrado ninguna coincidencia de plantilla para la vista "item_view" y el contenido "ajax".”*

Cuando hacemos clic en "Cargar más", no es de extrañar que esa llamada Ajax muestre los siguientes elementos de la receta. Podrías pensar que esto reutilizaría nuestra plantilla de vista de artículo "frontend", pero... en realidad hay una sección diferente específicamente para cuando el contenido se renderiza a través de Ajax. Copia por completo la sección `default` frontend, pégala y cámbiala por `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'
```

No hay que cambiar nada más: cuando estemos en modo `ajax`, utiliza la plantilla normal del frontend.

Ahora, si refrescamos la ruta Ajax... ¡funciona! Recarga la página de inicio y haz clic en "Cargar más". ¡Esto es muy bonito!

Traducir el botón de paginación

Aunque, cosa menor, nuestros diseñadores realmente quieren utilizar el texto "Mostrar más". No hay problema: todo lo que renderiza Layouts se procesa a través del traductor. Haz clic en el icono de traducción de la barra de herramientas de depuración web. ¡Ahí está! Al parecer, la clave de traducción es `collection.pager.load_more`.

Cópiala... y luego ve a abrir nuestro archivo de traducción - `nglayouts.en.yaml` - y pégalo. Mi editor cambió el formato... que en realidad funcionaría... pero volveré al formato más plano. Ponlo en "Mostrar más":

translations/nglayouts.en.yaml

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

Gira y... ¡lo tenemos!

Cambios CSS en la paginación

Vale, un cambio más para contentar a nuestros diseñadores. Inspecciona el elemento del botón. Layouts añade un montón de clases, que se estilizan a través del CSS que hemos incluido. Y, por supuesto, podemos anularlo si es necesario.

En nuestro editor, abre `assets/styles/app.css`. Como recordatorio, ya estamos ejecutando Webpack Encore en segundo plano. Así que, si cambiamos este archivo, ese cambio se reconstruirá automáticamente y se utilizará en el frontend.

En la parte inferior, pegaré algo de CSS para dar más margen a ese botón pero sin borde:

assets/styles/app.css

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

Volvemos a dar la vuelta, actualizamos y... nuestros diseñadores están contentos.

Así que, gracias a los diseños, obtenemos una paginación Ajax gratuita, que podemos personalizar con bastante facilidad. Eso es genial.

Rejillas frente a contenido Twig personalizado

Llegados a este punto, ya que somos capaces de renderizar rejillas y listas de recetas, podríamos entrar en el diseño "Lista de recetas" y sustituir este HTML codificado, que proviene de la plantilla: `templates/recipes/list.html.twig`. Sí, en teoría podríamos eliminar esto y sustituirlo por un bloque de lista.

El único problema... es que no se vería del todo bien. En lugar de renderizarse como lo hace ahora, Layouts utilizaría nuestra plantilla de artículos: así que cada artículo se vería como lo hace en la página de inicio.

Ahora, podemos arreglar esto creando una segunda forma de representar los elementos de la receta, y hablaremos de ello más adelante. Pero saco esto a colación por una razón importante. A no ser que pensemos reutilizar esta lista y su aspecto en otras páginas de

nuestro sitio, no hay grandes beneficios en hacer el trabajo de convertirla en algo que podamos representar mediante Diseños. Como sólo se utiliza aquí, renderizarla mediante Twig está perfectamente bien.

A continuación: vamos a mejorar el sistema de recetas haciendo posible la selección manual de elementos.

Chapter 11: Navegador de contenidos

Ahora podemos incrustar listas, cuadrículas o galerías de recetas en miniatura en cualquier diseño de forma dinámica. ¡Es genial! Y siempre podemos crear más tipos de consulta para, por ejemplo, elegir entre las últimas recetas o las recetas más populares.

Pero, ¿y si pudiéramos seleccionar recetas manualmente? Quizá queramos destacar cuatro recetas concretas en la página de inicio. En el área Diseños, en la rejilla, si cambiamos el "Tipo de colección", podemos cambiar a "Colección manual". Pero entonces... en realidad no podemos seleccionar ningún elemento.

Activar elementos manuales en la configuración

Para permitir que los elementos (en nuestro caso, las recetas) se seleccionen manualmente, primero tenemos que permitirlo en la configuración. Antes, cuando creamos la configuración `value_types`, pusimos `manual_items` en `false`. Cámbialo a `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
```

Y ahora, cuando intentamos acceder a la página, ¡nos aparece un error!

> *"El backend del Navegador de Contenidos Netgen para el tipo de valor `doctrine_recipe` no existe."*

¡Sí! Necesitamos implementar una clase que ayude a los Layouts a navegar por nuestras recetas. Eso se llama "navegador de contenido".

Configurar el "tipo de elemento" en NetgenContentBrowserBundle

En realidad, añadir un navegador de contenidos se hace mediante un bundle completamente distinto, que puedes utilizar fuera de Netgen Layouts. Es útil si necesitas una interfaz agradable para navegar y seleccionar elementos.

Como el navegador de contenidos se encuentra en un bundle diferente, no es necesario, pero voy a configurarlo con un nuevo archivo de configuración llamado `netgen_content_browser.yaml`. Dentro, establece la clave raíz en `netgen_content_browser` para configurar el "NetgenContentBrowserBundle":

config/packages/netgen_content_browser.yaml

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

Dentro de éste, podemos describir todas las diferentes "cosas manuales" que queremos poder navegar. Para ello, añade una clave `item_types` y, para el primer elemento, coge el nombre interno del tipo de valor - `doctrine_recipe` - para que coincidan, pégalo y dale un nombre. Qué te parece... `Recipes` con un bonito icono de fresa:

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
```

Lo único que necesitamos aquí es una clave `preview` con una subclave `template`, que pondré `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'
```

Y asegúrate de escribir "plantilla" correctamente. ¡Uy! De todas formas, estamos poniendo este `preview.template` porque la configuración nos lo exige... pero ya nos preocuparemos de crear esa plantilla más adelante.

Crear la clase backend

Si nos dirigimos y actualizamos... obtenemos el mismo error. Eso es porque necesitamos una clase backend que se conecte a este nuevo tipo de elemento. Crear un backend es un proceso sencillo, pero requiere algunas clases diferentes.

En el directorio `src/`, vamos a crear un nuevo directorio llamado `ContentBrowser/`... y dentro de él, una clase PHP llamada `RecipeBrowserBackend`. Ésta necesita implementar `BackendInterface`: la de `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 }
```

A continuación, ve a "Código"->"Generar" (o `Command`+`N` en un Mac) para implementar los nueve métodos que necesita No te preocupes: no es tan malo como parece:

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  }
```

Por último, para vincular esta clase backend al tipo de elemento en nuestra configuración, tenemos que dar a este servicio una etiqueta. Haremos esto de la misma forma que hicimos antes para el tipo de consulta: con `AutoconfigureTag`. De hecho, robaré este `AutoconfigureTag` ya que estoy aquí... pegaré eso... y añadiré la declaración `use` para ello. Esta vez, el nombre de la etiqueta es `netgen_content_browser.backend`, y en lugar de `type`, utiliza `item_type`. Ajústalo a la clave que tenemos en la config: `doctrine_recipe`. Pega y... ¡genial!

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  }
```

Esta vez cuando actualizamos... el error ha desaparecido. Añadamos temporalmente una nueva Rejilla al diseño... y elijamos "Colección manual". Ahora... ¡compruébalo! Como tenemos un backend, ¡vemos un botón "Añadir elementos"! Y cuando hacemos clic en él... falla. Eso no debería sorprendernos demasiado... ya que nuestra clase backend sigue estando completamente vacía. Si quieres ver el error exacto, puedes abrir la llamada AJAX.

Creación de la clase de ubicación

El sistema del navegador de contenidos funciona así: en estos métodos, describimos una "estructura de árbol", algo así como un sistema de archivos. las "ubicaciones" son como

directorios y los "elementos" son como los "archivos" o, en nuestro caso, las recetas individuales.

Vamos a simplificar mucho las cosas. En lugar de tener diferentes "directorios" o "categorías" de recetas por las que puedas navegar, vamos a tener un único directorio -o "ubicación"- en el que vivirán todas las recetas. Verás qué aspecto tiene esto en la interfaz de usuario dentro de unos minutos.

Para que esto funcione, dentro de `src/ContentBrowser/`, tenemos que crear una clase que represente una ubicación. La llamaré `BrowserRootLocation`. Esta clase... no es superinteresante: es sólo un poco de fontanería de bajo nivel que debemos tener. Haz que implemente `LocationInterface`, y a continuación, genera los tres métodos que necesitamos:

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.
    }
}
```

De nuevo, esta clase representará la única "ubicación". Así que para `getLocationId()`, podemos devolver cualquier cosa. Voy a `return 0`. Verás cómo se utiliza en un segundo. Para `getName()`, esto es lo que se mostrará en la sección de administración. Voy a `return 'All'`. Y para `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;
    }
}
```

Si tienes un sistema más complejo con múltiples subdirectorios, podrías crear una jerarquía de ubicaciones.

Muy bien, actualicemos nuestra clase backend para utilizar esto. Aquí arriba, `getSections()` será llamado en cuanto el usuario abra el navegador de contenidos. Nuestro trabajo consiste en devolver todos los "directorios" raíz, o "ubicaciones". Nosotros sólo tenemos uno: `return [new BrowserRootLocation()]`:

src/ContentBrowser/RecipeBrowserBackend.php

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

Después de llamar a éste, el navegador de contenidos llamará a `getLocationId()` en cada uno de ellos y hará una petición AJAX para obtener más información sobre ellos. En nuestro caso, esto ocurrirá una sola vez cuando el ID sea `0`. Parece raro, pero todo lo que tenemos que hacer es devolver esa misma ubicación: `if ($id === '0')`, y luego `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
}
```

Fíjate en que estoy utilizando `'0'` como cadena, pero... en `getLocationId()` devolvimos un número entero:

src/ContentBrowser/BrowserRootLocation.php

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

Eso es porque el id se pasará a JavaScript y se utilizará en una llamada Ajax. Para cuando llegue aquí, será una cadena. Un pequeño detalle a tener en cuenta.

Al final, por si acaso `throw` a `new \InvalidArgumentException()` y pasa un mensaje sobre una ubicación no válida:

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
}
```

¡Vale! Así que nuestro backend tiene una ubicación. Para los demás métodos, devolvamos lo más sencillo posible. Deja `loadItem()` vacío por un momento, para `getSubLocations()`, `return []`, para `getSubLocationsCount()`, `return 0`, para `getSubItems()`, `return []`, para `getSubItemsCount()`, `return 0`, para `search()`, `return []`... y finalmente, para `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;
    }
}
```

Uf... Hablaremos de cada uno de esos métodos más adelante. Pero nuestra clase backend ya es, al menos, algo funcional.

Si volvemos a actualizar el área de administración... hacemos clic en nuestra cuadrícula, y vamos a "Añadir elementos"... ¡se carga! ¡Di "hola" al navegador de contenido! Actualmente está vacío, pero puedes ver el "Todo", que es de nuestra única ubicación. Todavía no hay elementos dentro... porque tenemos que devolverlos desde `getSubItems()`. Hagámoslo a continuación

Chapter 12: Navegador de Contenidos: Devolver los elementos

Nuestro Navegador de Contenidos está funcionando más o menos. Podemos ver nuestra única ubicación... sólo que aún no tenemos ningún resultado. Esto se debe a que, para cualquier ubicación seleccionada, el Navegador de Contenidos llama a `getSubItems()`. Nuestro trabajo aquí es devolver los resultados. En este caso, todas nuestras recetas. Si tuviéramos varias ubicaciones, como recetas divididas en categorías, podríamos utilizar la variable `$location` para devolver el subconjunto. Pero haremos la consulta y devolveremos todas las recetas.

Consulta en getSubItems()

Para ello, ve a la parte superior de la clase y crea un constructor con `private RecipeRepository $recipeRepository`:

src/ContentBrowser/RecipeBrowserBackend.php

```
↕ // ... lines 1 - 4
5 use App\Repository\RecipeRepository;
↕ // ... lines 6 - 11
12 class RecipeBrowserBackend implements BackendInterface
13 {
14     public function __construct(private RecipeRepository $recipeRepository)
15     {
16     }
↕ // ... lines 17 - 70
71 }
```

Luego, aquí abajo en `getSubItems()`, di `$recipes = $this->recipeRepository` y utiliza el mismo método de antes: `->createQueryBuilderOrderedByNewest()`. A continuación añade `->setFirstResult($offset)`... y `->setMaxResults($limit)`. El Navegador de Contenidos viene con la paginación incorporada. Nos pasa el desplazamiento y el límite de la página en la que se encuentre el usuario, lo introducimos en la consulta y todos contentos. Termina con `getQuery()` y `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
}
```

Fíjate en que `getSubItems()` devuelve un `iterable`... en realidad se supone que es un iterable de algo llamado `ItemInterface`. Así que no podemos devolver simplemente estos objetos `Recipe`.

Crear la clase envolvente ItemInterface

En su lugar, en `src/ContentBrowser/`, crea otra clase llamada, qué tal `RecipeBrowserItem`. Haz que implemente `ItemInterface` -la de `Netgen\ContentBrowser` - y genera los cuatro métodos que necesita:

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.
    }
}
```

Esta clase será una pequeña envoltura de un objeto `Recipe`. Observa: añade un método `__construct()` con `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
}
```

Ahora, para `getValue()`, esto debería devolver el "identificador", así que `return $this->recipe->getId()`. Para `getName()`, sólo necesitamos algo visual que

podamos mostrar, como `$this->recipe->getName()`. Y para `isVisible()`, `return true`. Esto es útil si un `Recipe` puede estar publicado o no. Tenemos una situación similar con `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;
    }
}
```

Si tuvieras un conjunto de reglas en las que quisieras mostrar ciertas recetas pero hacer que no se pudieran seleccionar, podrías `return false` aquí.

Y... ¡ya está! ¡Ha sido fácil!

De vuelta a nuestra clase backend, necesitamos convertir estos objetos `Recipe` en objetos `RecipeBrowserItem`. Podemos hacerlo con `array_map()`. Volveré a utilizar la elegante sintaxis `fn()`, que recibirá un argumento `Recipe $recipe`, seguido de `=> new RecipeBrowserItem($recipe)`. Para el segundo arg, pasa `$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
}
```

Es una forma elegante de decir

> *"Recorre todas las recetas del sistema, crea un nuevo `RecipeBrowserItem` para cada una, y devuelve esa nueva matriz de elementos."*

Muy bien, ¡vamos a ver qué aspecto tiene! Actualiza el diseño, haz clic en la Rejilla, vuelve a "Añadir elementos" y... ¡ya está! ¡Vemos diez elementos!

Implementando getSubItemsCount()

Pero deberíamos tener varias páginas. Ah, eso es porque seguimos devolviendo `0` desde `getSubItemsCount()`. Vamos a arreglarlo. Roba la consulta de arriba... pega, devuelve esto, quita `setFirstResult()` y `setMaxResults()`, añade `->select('COUNT(recipe.id)')`, y luego llama a `getSingleScalarResult()` al final:

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
}
```

Y así, cuando actualicemos... y abramos el Navegador de Contenidos... ¡tendremos páginas!

Añadir la función de búsqueda

Vale, pero ¿podemos buscar recetas? Por supuesto. Podemos aprovechar `search()` y `searchCount()`. Esto es muy sencillo. Roba toda la lógica de `getSubItems()`, pégala en `search()` y pasa `$searchText` al método QueryBuilder, que ya permite este argumento:

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
}
```

Si quieres tener un poco menos de duplicación de código, podrías aislar esto en un método `private` en la parte inferior.

Copia también la lógica del otro método de recuento... pégalo en `searchCount()`, y pásalo también a `$searchText`:

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();
    }
}
```

Y así de fácil, si nos movemos hacia aquí e intentamos buscar... funciona. ¡Estupendo!

Muy bien - selecciona algunos elementos, pulsa "Confirmar" y... ¡oh no! ¡Se rompe! Sigue diciendo "Cargando". Si miras hacia abajo en la barra de herramientas de depuración web, tenemos un error 400. Maldita sea. Cuando lo abrimos, vemos

> *"El cargador de valores para el tipo de valor `doctrine_recipe` no existe."*

Sólo nos falta una pieza final: Una clase muy sencilla llamada "cargador de valores". Eso a continuación.

Chapter 13: Cargador de valores + Plantilla de vista previa

Así que nuestro navegador de contenidos funcionaba de maravilla... hasta que seleccionamos un elemento. En ese momento, eligió hacer una cosa extraña: ¡explotar! La llamada Ajax que falló dice

> *"El cargador de valores para el tipo de valor `doctrine_recipe` no existe."*

Para repasar: tenemos un tipo de valor personalizado llamado `doctrine_recipe`, que creamos para poder añadir cuadrículas y listas de entidades `Recipe`. Para que esto funcione, tenemos (1): un conversor de valores para convertir los objetos `Recipe` a un formato que entiendan los diseños. (2) una clase de consulta que nos permita utilizar colecciones dinámicas. (3) una clase de backend de navegador para permitirnos seleccionar elementos manuales. Y ahora (4), necesitamos un cargador de valores que sea capaz de tomar el "id" de estos elementos seleccionados manualmente y convertirlos en objetos `Recipe`. Ésta será la última "cosa" que necesitaremos para nuestro tipo de valor, ¡lo prometo!

Crear y etiquetar el cargador de valores

Dentro del directorio `src/Layouts/`, crea una nueva clase llamada `RecipeValueLoader`, haz que implemente `ValueLoaderInterface` y genera los dos métodos que necesita:

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.
    }
}
```

Son bastante sencillos. Pero, antes de rellenarlos, vuelve a la ruta Ajax y actualiza para ver... exactamente el mismo error. ¿Por qué? Como hemos visto con otras cosas, necesitamos "asociar" este `RecipeValueLoader` a nuestro tipo de valor `doctrine_recipe`. ¿Cómo? Sin sorpresas Con una etiqueta. Digamos `#[AutoconfigureTag()]` y esta vez se llama `netgen_layouts.cms_value_loader`. Para el segundo argumento, pasa `value_type` ajustado a `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! Si recargamos ahora... ¡mejor! Ese error se debe a que aún no hemos rellenado la lógica.

Añadir la lógica

Muy sencillo, necesitamos tomar el ID y devolver el objeto `Recipe`. Para ello, crea un constructor que acepte un argumento `RecipeRepository $recipeRepository`. Y... déjame limpiar las cosas:

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
}
```

Ahora, aquí abajo, devuelve `$this->recipeRepository->find()` y pasa `$id`. Para `loadByRemoteId()`, que sólo necesitamos si vamos a utilizar la función de importación para mover contenido entre bases de datos, sólo `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);
    }
}
```

Y ahora... ¡la llamada Ajax funciona! Y lo que es más importante, si actualizamos todo el administrador de diseños... ¡sí! ¡Mira nuestra cuadrícula! ¡Tenemos cuatro elementos manuales! ¡Eso es genial! Podemos reordenarlos si queremos, añadir más, eliminarlos, lo que sea.

Prueba a publicar esta página y luego recarga la página de inicio. ¡Ahí están! Aunque faltan nuestras "últimas recetas". ¡Vaya! Creo que accidentalmente también cambié esto a una

colección manual. Vuelve a cambiarla a una colección dinámica, se ve bien, publica y.... ahora... genial: todo está de vuelta.

Añadir la vista previa

Así que ya podemos seleccionar elementos manuales a través del navegador de contenido... aunque cuando añadimos originalmente la configuración para todo esto, establecimos una plantilla de vista previa... ¡pero nunca la creamos!

Abramos de nuevo el navegador de contenidos. Entonces, en la rejilla manual, pulsa "Añadir elementos". La plantilla de vista previa activa el modo de vista previa aquí arriba. Si hacemos clic en un elemento, nos muestra una vista previa. Bueno, lo haría... excepto porque en realidad no hemos añadido esa plantilla.

Para que esto funcione, tenemos que hacer dos pequeñas cosas. Primero, abrir `RecipeBrowserBackend`. Aquí nos hemos saltado algunos métodos. Por ejemplo, omitimos `getSubLocations()` y `getSubLocationsCount()` porque sólo son necesarios si tienes una jerarquía de ubicaciones.

También nos hemos saltado `loadItem()`. Se utiliza para la vista previa. Nos pasará el ID de lo que se ha cargado y necesitamos devolver un `ItemInterface`. Así de sencillo, podemos devolver un `new RecipeBrowserItem()` -que es la pequeña clase que envuelve al `Recipe` - pasando `$this->recipeRepository->find($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 }
```

¡Genial! Lo único que tenemos que hacer es... ¡crear la plantilla de vista previa! En `templates/nglayouts/`, añade un nuevo directorio llamado `content_browser/`, y dentro, un nuevo archivo llamado `recipe_preview.html.twig`. Para empezar, sólo tienes que imprimir la función `dump()`:

templates/nglayouts/content_browser/recipe_preview.html.twig

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

Lo bueno es que ni siquiera necesitamos actualizar. Mientras hagamos clic en un elemento en el que no hayamos hecho ya clic... ¡funciona! Y mira esta variable `item`: es una instancia de `RecipeBrowserItem`... así que una instancia de esta clase de aquí.

Eso es genial... excepto que `RecipeBrowserItem` no tiene una forma de que obtengamos el objeto `Recipe` real. Afortunadamente, podemos arreglar eso nosotros mismos. Después de todo, ¡esta es nuestra clase! Iré a "Código"->"Generar" para generar un método `getRecipe()`:

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 }
```

Ahora, en la plantilla, podemos decir `{{ item.recipe.name }}`. Y para hacerlo más elegante, añade un `<img` cuyo `src` se establezca en `item.recipe.imageUrl`... también con un atributo `alt`:

templates/nglayouts/content_browser/recipe_preview.html.twig

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

Una vez más, no necesitamos actualizar. Si haces clic en un elemento que ya has previsualizado, lo cargará desde la memoria. Pero si haces clic en uno nuevo... ¡sí! ¡Ahí está nuestra vista previa! Genial.

Vale, ya hemos terminado con los elementos manuales, el navegador de contenidos y todo esto. Por cierto, hay una forma de añadir más columnas a esta tabla, como nombre de archivo, tamaño de archivo, fecha de creación, etc. No vamos a hablar de eso, pero es totalmente posible.

Comprobación de estado: en este punto, tenemos la capacidad de añadir un diseño a cualquier página, reordenar el contenido de la página, añadir título, texto, bloques HTML, o incluso listas

y cuadrículas de recetas dinámicas. Eso es mucho poder. ¡Ahora quiero más poder! Quiero que sea posible utilizar la cuadrícula y los bloques de lista para añadir otros elementos a nuestra página... elementos que no viven en absoluto en nuestra base de datos. Eso a continuación.

Chapter 14: Contentful: Cargar datos de un CMS externo

Si añadiéramos cinco entidades más y quisiéramos poder seleccionarlas como elementos en el admin de Layouts, podríamos añadir otros cinco tipos de valor, tipos de consulta y vistas de elementos. Ahora que sabemos lo que estamos haciendo, es un proceso bastante rápido y nos daría mucha potencia en nuestro sitio.

Pero una de las cosas bonitas de los Layouts es que nuestros tipos de valor pueden proceder de cualquier parte: una Entidad Doctrine, datos de una API externa, datos de un almacén Sylius o de Ibexa CMS. De hecho, sistemas como Sylius e Ibexa ya tienen paquetes que hacen todo el trabajo de integrar y añadir los tipos de valor por ti.

Una de las grandes piezas que faltan en nuestro sitio son las habilidades. Las habilidades de la página de inicio están codificadas y el enlace "Todas las habilidades" ni siquiera va a ninguna parte. Podríamos haber optado por almacenar estas habilidades localmente a través de otra entidad Doctrine, pero en lugar de eso, vamos a cargarlas desde una API externa a través de un servicio llamado "Contentful".

¡Hola Contentful!

Me dirigiré a Contentful.com e iniciaré sesión. Esto me lleva a un espacio "Contentful" llamado "Bark & Bake" que ya he creado. ¡Contentful es increíble! Es básicamente un CMS como servicio. Nos permite crear diferentes tipos de contenido llamados "modelos de contenido". Ahora mismo, tengo un modelo de contenido llamado "Habilidad" y otro llamado "Anuncio". Si hiciéramos clic en ellos, podríamos introducir contenido a través de una interfaz superamigable. Ya he creado 5 habilidades, cada una con un montón de datos.

Así que, aquí creas y mantienes tu contenido. Luego Contentful tiene una API restful que podemos utilizar para obtener todo esto.

Contentful es genial. Pero el objetivo de esto no es enseñarte sobre Contentful, ¡no! Se trata de mostrarte cómo podemos obtener contenido para los diseños desde cualquier lugar. Por ejemplo, si queremos cargar "habilidades" de Contentful, podríamos crear manualmente un

nuevo tipo de valor y hacer todo el trabajo que hicimos antes, excepto hacer peticiones a la API de Contentful en lugar de consultar la base de datos.

¡Pero! ¡Ni siquiera necesitamos hacer eso! ¿Por qué? Porque Layouts ya tiene un bundle compatible con Contentful. Ese bundle añade el tipo de valor, algunos tipos de consulta, las vistas de elementos e incluso la integración del navegador de contenido por nosotros. Woh.

¡Vamos a cogerlo!

Instalar el bundle Contentful

Ve a tu terminal y ejecuta:

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

El `-W` está ahí sólo porque, al menos al grabar esto, Composer necesita poder degradar un pequeño paquete para contentar a todas las dependencias. Esa bandera le permite hacerlo.

De acuerdo La receta de este paquete ha añadido un nuevo archivo de configuración: `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)%"
```

Y éste lee dos nuevas variables de entorno... que viven en `.env`:

.env

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

Ya que estamos aquí, actualicemos estos valores para que apunten a mi espacio Contentful. Copia las claves del bloque de código de esta página y pégalas aquí. Aquí están mi`CONTENTFUL_SPACE_ID`... y mi `CONTENTFUL_ACCESS_TOKEN`, que nos darán acceso de lectura a mi espacio:

.env

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

Contentful + Layouts

Vale, la integración Layouts + Contentful nos da dos cosas muy distintas, y es súper importante entender la diferencia para que todo quede claro.

En primer lugar, el paquete añade una integración entre Layouts y Contentful. Esto significa que añade nuevos tipos de valores, nuevos tipos de consulta y todas las demás cosas que acabamos de añadir para Doctrine. En otras palabras, podemos añadir instantáneamente las competencias o anuncios de Contentful en bloques de lista o cuadrícula. Eso es genial, y lo veremos pronto.

La segunda cosa que añade la integración de Contentful no tiene nada que ver con Layouts. Son las rutas dinámicas. Añade un sistema para que cada "elemento" de Contentful esté disponible a través de su propia URL. Literalmente, todas estas habilidades tendrán al instante su propia página en nuestro sitio. Esto no tiene nada que ver con los Diseños, que consisten en controlar el diseño de las páginas existentes en tu sitio, no en añadir páginas nuevas.

Configurar el enrutamiento dinámico

Pero, como Contentful es un CMS, es bueno tener una página para cada contenido. Para poner en funcionamiento las rutas dinámicas, entra en el directorio `config/packages/` y añade un nuevo archivo llamado `cmf_routing.yaml`. CMF Routing es un paquete que Contentful utiliza entre bastidores para añadir las rutas dinámicas. Pego aquí un poco de configuración:

config/packages/cmf_routing.yaml

```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
```

Es feo... pero esta parte no tiene nada que ver con Layouts, así que no importa demasiado. Se trata de permitir que Contentful añada automáticamente URL dinámicas a nuestro sitio.

Este sistema de enrutamiento almacena las rutas en la base de datos... y eso significa que necesitamos una nueva base de datos. Dirígete a tu consola y ejecuta:

```
symfony console make:migration
```

Y... Me aparece un error. Grosero. Probemos a borrar la caché... puede que haya pasado algo raro... o que aún no haya visto mi nuevo archivo de configuración.

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

Una vez borrada la caché... Volveré a hacer la migración:

```
symfony console make:migration
```

Esta vez... ¡perfecto! Abro el directorio `migrations/`, busco ese archivo y... ¡se ve bien!

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  }
```

Tenemos unas cuantas tablas que contienen información sobre nuestros datos de Contentful... y unas cuantas para almacenar esas rutas dinámicas.

Ahora Ejecuta:

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

Y... ¡woohoo! Tenemos las nuevas tablas que necesitamos.

Por último, podemos ejecutar un comando para cargar todo nuestro contenido desde Contentful y crear esas rutas dinámicas. Una vez más, se trata de una funcionalidad que no tiene nada que ver con los Diseños. Ejecuta:

```
symfony console contentful:sync
```

Y... ¡precioso! Cargó seis elementos. En producción puedes configurar un webhook para que tu sitio se sincronice instantáneamente con cualquier cambio que hagas en Contentful. Pero mientras estamos desarrollando, ejecutar este comando funciona bien.

El resultado de este comando es que cada contenido de Contentful tiene ahora su propia página Para verlas, ejecuta:

```
symfony console contentful:routes
```

Y... ¡impresionante! Parece que tengo una URL llamada `/mashing`. Vamos a comprobarlo! Vuelve a nuestro sitio, navega hasta `/mashing` y... ¡funciona! Más o menos. Está aquí, pero la parte central está vacía.

Hablemos de lo que ocurre a continuación y de cómo podemos aprovechar los Diseños para dar vida a esta página.

Chapter 15: Asignar un diseño a las páginas de Contentful

La integración de Contentful que acabamos de instalar ha añadido dos cosas a nuestro sitio. En primer lugar, ha añadido una integración de Layouts: nuevos tipos de valores, tipos de consulta, etc. para que podamos seleccionar nuestro contenido de Contentful en bloques de listas y cuadrículas. En segundo lugar, ha añadido la posibilidad de que cada contenido de Contentful tenga su propia URL y página en nuestro sitio. La segunda parte no tiene nada que ver con los Diseños.

Hace un minuto, utilizamos este práctico comando `contentful:routes` para ver que ahora debería haber una página en la URL `/mashing`. Cuando fuimos allí, no nos dio un error 404, pero no funcionó exactamente. La página está casi vacía.

Depuración del funcionamiento de las páginas dinámicas de Contentful

Veamos qué está pasando. Haz clic en el icono Twig de la barra de herramientas de depuración web para averiguar qué plantillas se están renderizando. Veamos aquí... si bajamos un poco... aparentemente se renderiza `@NetgenLayoutsContentful/contentful/content.html.twig`. ¡Esa debe ser la plantilla de esta página! Vamos a comprobarlo.

Le daré a `Shift`+`Shift` y buscaré `content.html.twig`: queremos la de `layouts-contentful`. Y... ¡genial! Esta es la plantilla que está renderizando esa página. Imprime `content.name`... pero en realidad nunca la vemos. Ah, eso es porque la renderiza en un `block` llamado `content`. Éste acaba extendiendo `base.html.twig`... y como nuestra plantilla base nunca renderiza `block content`, no vemos nada. De nuevo, esta parte de Contentful en la que obtienes una URL que renderiza un controlador, que a su vez renderiza esta plantilla... no tiene nada que ver con los Diseños. Es sólo una bonita forma de exponer cada pieza de contenido de Contentful como una página de nuestro sitio.

Así que, sin relación con Layouts, si quisiéramos, podríamos reemplazar esta plantilla en nuestra aplicación y personalizarla para que funcione. Podríamos cambiarla para utilizar

`block body` y aprovechar esta variable `content`, que representa el contenido, para mostrar todos los campos.

Pero... espera un segundo. ¿No es ese el objetivo de Layouts? Los diseños nos permiten crear páginas dinámicamente, en lugar de escribirlas completamente en Twig. Ahora mismo, esta página no está vinculada a ningún diseño. Pero si la vinculáramos, podríamos empezar a construir la página utilizando los datos de la Habilidad Contentful correspondiente, en este caso, de la Habilidad "Machacar".

Asignar un diseño a la página dinámica

Dirígete a nuestra sección de administración, publica ese diseño y crea un nuevo diseño. Lo llamaré "Diseño de habilidad individual"... y elige "Diseño 2". Con el tiempo, haremos que se parezca más al "Diseño 5"... pero podemos hacerlo más adelante mediante los bloques de columnas. Ésa es una de las razones por las que me gusta la "Presentación 2": es bastante sencilla, y podemos hacerla más compleja más adelante con las herramientas que ya tenemos.

Bien, empieza como siempre. Cierra la barra de herramientas de depuración web para que podamos vincular la cabecera a la cabecera compartida... y nuestro pie de página al pie de página compartido. Estupendo. Luego, para empezar, añade un bloque Título, escribe algo... y publica el diseño.

Asignar un diseño a las entradas de Contentful

A continuación, tenemos que asignar este diseño a esa página. Hasta ahora, hemos mapeado maquetaciones por el nombre de la ruta o por la URL, también conocida como "Información de la ruta". Podríamos volver a hacerlo aquí. Pero, como verás, lo que realmente queremos hacer es utilizar este diseño para todas las páginas de Skills. Dentro de unos minutos, cambiaremos la URL de estas páginas de algo como `/mashing` a `/skills/mashing`. Cuando lo hagamos (permíteme añadir una nueva asignación aquí y pulsar detalles), podríamos utilizar el "Prefijo de información de ruta" para asignar esta disposición a cualquier URL que empiece por `/skills/`.

Pero, algo que puede añadirse a los Diseños es otra forma de mapear o resolver qué diseño debe utilizarse en cada página. Y, ¡sí! El bundle Contentful añadió dos nuevos: Entrada Contentful y Espacio Contentful. Cuando vamos a una de estas páginas de Contentful, la ruta

dinámica le dice a Symfony a qué Contentful Entry - que es la pieza individual de contenido en Contentful - y a qué Contentful Space corresponde esta página.

Gracias a esto, podemos aprovechar uno de estos nuevos objetivos para que coincida con la entrada o el espacio. Por ejemplo, podríamos utilizar la Entrada Contentful para mapear un diseño específico a un elemento específico en Contentful. Literalmente, podríamos decir

> *"Si el Contenido actual es específicamente esta habilidad "Machacar", entonces utiliza esta disposición."*

O podríamos hacer lo que yo voy a hacer: mapear a través del Espacio de Contentful. Sólo tenemos un Espacio, así que es bastante fácil. Básicamente, estamos diciendo

> *"Si estamos en cualquier página dinámica de Contentful, quiero que mapees a este diseño."*

Guardemos esto... y luego vinculemos este diseño al "Diseño de habilidad individual". Pulsa "Confirmar" y... ¡listo! Ve, actualiza y... ¡funciona! ¡Sí!

Asignación a un tipo de contenido específico

Como he mencionado antes, en realidad tenemos dos tipos de contenido en Contentful: Habilidades y Anuncios. Los anuncios no deben tener su propia página, sólo las habilidades. Vamos a incrustar anuncios en algunas páginas existentes un poco más adelante.

Vuelve a los detalles de la vinculación del diseño. Además del espacio Contentful, podemos bajar aquí a una lista de condiciones y seleccionar "Tipos de contenido Contentful". Las condiciones son una forma de hacer que tu vinculación sea más específica. Añade esa condición. Y, esto es un poco difícil de ver, pero podemos seleccionar "Habilidad" o "Anuncio". Selecciona "Habilidad", guarda los cambios y... ¡genial! Ahora sólo coincidirá si vamos a una URL de Contentful que esté mostrando una habilidad.

En la línea de comandos, puedes ver que tenemos un anuncio... es esta URL de aspecto gracioso. Sí, ahora mismo, el anuncio está disponible como página en nuestro sitio. Lo arreglaremos en unos minutos. Pero, como mínimo, si fuéramos a esa URL rara, la página funcionaría... pero no coincidiría con ningún diseño gracias a nuestro mapeo. Así que, básicamente, estaría en blanco.

Así que ahora tenemos control sobre las páginas de Contentful. ¡Genial! Aunque... lo único que estamos renderizando es un título manual. Snooze.

A continuación: Hagamos nuestro diseño más inteligente mostrando contenido real de la habilidad correspondiente.

Chapter 16: Construir la página Contentful

Ahora tenemos un control total sobre cómo se muestran las páginas de Contentful. Eso es gracias al "diseño de habilidad individual" que hemos asignado a todas las páginas de "habilidad" de Contentful.

Pero... todo lo que tenemos es este título manual de `h1`. ¿Cómo podemos mostrar los datos reales de la habilidad de Contentful que estamos viendo?

En primer lugar, en el sitio de Contentful, si navego hasta "Modelo de contenido" y hago clic en "Skill", puedes ver que cada Skill tiene 5 campos... y cada campo tiene un nombre interno. Es... casi más fácil ver esto a través de la vista previa JSON. Allá vamos. Así que hay un campo "Título", su nombre interno es `title`, "Descripción breve", "Técnica", y algunos otros como "Imagen" y "Anuncio". El anuncio es en realidad un enlace a ese otro tipo de contenido.

Uso del tipo de bloque "Campo de entrada de contenido

En cualquier caso, lo que realmente queremos hacer aquí es imprimir el título de la habilidad en `h1`. Afortunadamente, eso es posible, gracias a un nuevo tipo de bloque que ha añadido el bundle Contentful. Está aquí abajo: "Campo de entrada Contentful".

Esto nos permite renderizar un único campo de cualquier entrada de Contentful que se esté renderizando en ese momento. ¡Vamos a probarlo! A continuación, borra el antiguo `h1`.

El nuevo bloque tiene una opción superimportante: identificador de campo. Establécelo con el nombre interno del campo: `title`. Y conviértelo en un `h1`. Como de costumbre, la etiqueta del bloque es opcional, pero yo la incluiré.

¡Genial! Dale a publicar y sigue editando, muévete y... ¡sí! Es dinámico. Si vamos a la URL de alguna otra habilidad, como `/basic-chop`, ¡también funciona!

Añadir el área de héroe

Pongámonos más elegantes. Añade una columna... y mueve este título dentro. ¿Adivinas lo que voy a hacer? Dale a la columna la misma clase `hero-wrapper` que hemos utilizado antes. ¿Y sabes qué más? Cada habilidad tiene una "Breve descripción". Vamos a añadir otro bloque de campo de entrada justo debajo.

Fíjate en que una opción de este bloque es "tipo de vista". Pronto hablaremos más sobre esto, pero debería coincidir con el "tipo" del contenido que estás extrayendo de Contentful. Hasta ahora, tanto `title` como este `shortDescription` son tipos "cadena". Deja esto como `div`.

¡Temporizador de pruebas! Pulsa "Publicar y continuar editando". Y... a ver qué tal queda. ¡Me encanta! ¡Añadamos más!

Añadir una imagen de Contentful

Cada habilidad tiene una imagen. Dentro de esa misma columna héroe, añade otro bloque de entrada Contentful en la parte inferior. Se llamará `image`... y el tipo será "activos referenciados". Tienes que establecer una anchura y una altura. Hagamos 200 por 200. Publica eso... actualiza y... ¡ya estamos en marcha!

Una última cosa: renderizar el contenido de la habilidad debajo de todo. Por cierto, podríamos renderizarlo en la misma zona... o utilizar la zona de abajo. Las zonas no importan mucho en la mayoría de los casos.

Utilizar un bloque de 2 columnas

Pero hagamos este punto más interesante. Quiero mostrar el contenido de la habilidad a la izquierda y un anuncio a la derecha. Para ello, utiliza por primera vez un bloque de 2 columnas. Ajústalo a 66, 33 para que el lado izquierdo ocupe la mayor parte del espacio. Añade un título a la parte izquierda y conviértelo en un `h3` con el texto "La Técnica:". Debajo, arrastra un campo de entrada de contenido.

Éste... si voy a comprobar mis campos... se llama `technique` y contiene texto enriquecido. Si lo modificaras en Contentful, verías un editor de texto enriquecido... y el valor final es HTML. Así que escribe `technique`, mantenlo como `div` y selecciona `Richtext`.

Renderizar una entrada relacionada de Contentful

Por último, en el lado derecho, añade un campo de entrada Contentful más. Vuelve a mirar el modelo de contenido de Habilidades... y desplázate un poco hacia abajo. El que queremos utilizar se llama `advertisement`, y es de tipo "Entrada referenciada". Sí, si editaras una habilidad, elegirías el Anuncio de la lista de Anuncios que tenemos en Contentful. Es como una relación de base de datos.

De todos modos, introduce `advertisement`, pulsa "Publicar y continuar editando"... actualiza y... ¡bien! Más o menos genial. Necesitamos un contenedor para introducirlos. Ya tenemos una columna, así que haz clic en "Envolver en contenedor".

Y... sí... aunque a esto también le vendría bien algo de margen superior. En esa misma columna, añade una clase: `my-3`. Publica esto... y vuelve a cargar. ¡Mucho mejor! Aunque, el Anuncio sólo está imprimiendo una URL... no renderizando un anuncio. Eso es porque Contentful no sabe cómo renderizar el tipo de contenido "Anuncio". Pronto lo solucionaremos.

Pero primero, vamos a arreglar nuestras páginas Skill anteponiendo a todas las URL el prefijo `/skills`.

Chapter 17: Personalizar el Slugger de Contentful

Antes de que sigamos personalizando el aspecto de nuestro sitio, quiero arreglar las URL de las habilidades para que en lugar de ser `/mashing`, la página sea `/skills/mashing`. Recuerda: el hecho de que nuestro contenido Contentful tenga instantáneamente URL en nuestro sitio proviene del paquete Contentful que instalamos antes. Pero esa magia no tiene nada que ver con los Diseños. Por tanto, personalizar esta URL también es específico de Contentful, no de Layouts. Pero... Realmente quiero arreglarlo.

Crear la clase Slugger

En el directorio `src/Layouts/`, crea una nueva clase llamada `ContentfulSlugger`. Haz que implemente `EntrySluggerInterface`... y genera el único método que necesitamos: `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.
    }
}
```

Vamos a configurar las cosas para que se llame a este método cuando se creen las URL dinámicas de todas las entradas de Contentful. Nos permitirá controlar el "slug", que en realidad es la URL de cada elemento.

Para facilitarte las cosas, utiliza `FilterSlugTrait` para acceder a un método que utilizaremos dentro de un minuto:

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
}
```

Vale, en Contentful tenemos tanto Habilidades como Anuncios. Pero en realidad no queremos que los anuncios tengan su propia página. Por desgracia, con la integración de Contentful, no hay forma de desactivar las URL para un tipo de contenido específico. Hablaré de cómo solucionarlo en un minuto.

En cualquier caso, este método se pasará tanto a las habilidades como a los anuncios. Utiliza la nueva función PHP `match()` para que coincida con `$contentfulEntry->getContentType()->getId()`. Eso devolverá el nombre interno de cada tipo, que puedes encontrar en Contentful. Si es `skill`, devuelve `/skills/` y luego `$this->filtersSlug()` -que viene del rasgo- pasando `$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
        };
    }
}
```

Para `advertisement`, devuelve `/_ad` para todos ellos:

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
        };
    }
}
```

Al menos, en este punto, sólo un anuncio podría tener una página en nuestro sitio: si el usuario fuera a `/_ad`, coincidiría con el primero.

Al final, lanza una nueva Excepción con "Tipo no válido":

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'),
        };
    }
}
```

Así que, sí, en este punto, los anuncios seguirán teniendo su propia página. No hay forma de desactivar eso desde el principio. Pero si te importa lo suficiente, yo asignaría todos los anuncios a la misma URL o patrón de URL de esta forma. Luego crearía una ruta y un controlador con la misma URL y devolvería un 404. Esa ruta tendrá prioridad sobre la dinámica.

Etiquetar y configurar el Slugger

Para decirle a Contentful que utilice nuestro slugger, necesitamos, por supuesto, ¡darle una etiqueta! Añade `#[AutoconfigureTag]` y éste se llamará `netgen_layouts.contentful.entry_slugger`. Esto también necesita una opción `type`... que puedes establecer en cualquier cadena. Utilicemos `default_slugger`:

src/Layouts/ContentfulSlugger.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\DependencyInjection\Attribute\AutoconfigureTag;
9
10 #[AutoconfigureTag('netgen_layouts.contentful.entry_slugger', ['type' =>
   'default_slugger'])]
11 class ContentfulSlugger implements EntrySluggerInterface
12 {
↕ // ... lines 13 - 22
23 }
```

¿Cómo se utiliza? En `config/packages/`, necesitamos crear un nuevo archivo de configuración para el paquete contentful de layouts. Llamémoslo `netgen_layouts_contentful.yaml`.

Repite eso para la clave raíz. A continuación, añade `entry_slug_type`, luego `default` configurado con el tipo que hemos utilizado en nuestra etiqueta: `default_slugger`:

config/packages/netgen_layouts_contentful.yaml

```
1 netgen_layouts_contentful:
2     entry_slug_type:
3         default: default_slugger
```

Esta curiosa sintaxis dice

> *"Para cada tipo de contenido en Contentful, utiliza `default_slugger` al generar la URL. Por lo tanto, utiliza nuestro `ContentfulSlugger`."*

Vale, ¡listo! Pero... esto no se llama cuando recargamos la página. No. Se llama cuando "sincronizamos" nuestro contenido desde Contentful. Bien, ¡vamos a resincronizar! En tu terminal, ejecuta:

```
symfony console contentful:sync
```

Esto actualiza nuestra base de datos local con los últimos datos de Contentful... y funcionó bien. Pero cuando ejecutamos:

```
symfony console contentful:routes
```

¡Las URL no cambiaron! Esto es una peculiaridad... o quizás una característica para que las páginas existentes no se rompan. En cualquier caso, una vez que se importa una ruta por primera vez, su URL nunca cambia.

La forma más fácil de restablecer las cosas es eliminar la tabla de rutas y volver a importar todo.

Y esto es bastante divertido. Podemos Ejecuta:

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

Eso anulará la migración más reciente, haciendo que se eliminen las tablas contentful y de rutas. Vuelve a ponerlas con:

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

Vuelve a sincronizar el contenido:

```
symfony console contentful:sync
```

Y ahora comprueba las rutas:

```
symfony console contentful:routes
```

¡Sí! ¡La URL es `/skills/mashing`! Así, en `/mashing`, obtenemos un 404 a la antigua usanza. Pero `/skills/mashing` funciona.

Siguiente: aún no tenemos una página que enumere todas las habilidades. ¡Vamos a arreglarlo!

Chapter 18: La página de lista de habilidades + una cuadrícula de habilidades

Gracias a la integración con Contentful, además de nuestro tipo de valor `doctrine_recipe`, ahora tenemos un segundo tipo de valor que puede cargar cosas de Contentful. Esto significa que podemos mostrar listas y cuadrículas de habilidades dentro de cualquier diseño, como aquí en nuestra página de inicio.

¡Vamos a probarlo! Publica este diseño... y edita el Diseño de la Página de Inicio. Ah, y podemos eliminar esta antigua cuadrícula con la que estábamos jugando antes.

Abajo, estamos representando el bloque Twig de `featured_skills`. Pero en realidad, si miras nuestra plantilla, ¡están totalmente codificados!

Añadir una cuadrícula de habilidades

¡No hay problema! Añade un bloque Rejilla... que ya está configurado como "Colección manual" ¡Pero mira esto! ¡Ahora podemos elegir entre seleccionar manualmente "Entradas de contenido" o recetas! Y cuando hacemos clic en "Añadir elementos", ¡el navegador de contenidos ya funciona!

Selecciona unos cuantos... bien... luego publica esto. Actualiza. Um... ¡bien! Sí se muestran... pero sólo el título. Buen comienzo. Para hacerlo un poco mejor, ve a la pestaña "Diseño"... y envuelve esto en un contenedor.

Eso debería, al menos, darnos algunos canalones. Ya está. En última instancia, queremos que se muestren como las habilidades codificadas que hay debajo. Y vamos a trabajar en ello en unos minutos.

Añadir una página /habilidades

Pero antes de llegar ahí, ¿qué tal una página `/skills` que enumere todas las habilidades? Bueno, la integración de Contentful no nos dio esta URL. Pero, ¡no hay problema! ¡Podemos

crearla nosotros mismos en Symfony!

Bueno, en realidad, ¡podríamos hacerlo completamente en Contentful! Podríamos crear un tipo de contenido "Página", crear una página "Habilidades", que podría convertirse en `/skills`, y luego asignarla a un Diseño. Este es el tipo de cosas que harías normalmente cuando tienes un CMS a tu alcance

Pero crearemos esta página de forma manual. Al fin y al cabo, los Diseños en realidad sirven para ayudar a organizar el aspecto de las páginas existentes... en realidad no se trata de añadir páginas dinámicas. Ése es un trabajo para un CMS.

En tu editor, abre `src/Controller/MainController.php`. Copia la acción `homepage()`, pégala, cámbiala a `/skills`, llámala `app_skills` y renombra el método a `skills()`. Para la plantilla, renderiza `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 }
```

Ahora, en el directorio `templates/main/`, crea esto: `skills.html.twig`. Empecemos por lo más pequeño posible: extender `nglayouts.layoutTemplate`:

templates/main/skills.html.twig

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

Genial. Ya que estamos aquí, abre `base.html.twig` y enlaza con esto. Busca "Habilidades", ahí está el enlace. Establece el `href` en `{{ 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>
```

¡Me gusta! Actualiza, prueba el enlace de la cabecera y... ¡la página funciona!

¿Añadir contenido manualmente?

Para poner contenido en esta página, ¡también podríamos hacerlo manualmente escribiendo código en nuestra app! La biblioteca Contentful que instalamos antes tiene un servicio `ClientInterface` que podríamos utilizar para obtener todas estas competencias de Contentful en nuestro controlador.

Pero en lugar de eso, vamos a tomar el camino más fácil y dejar que los diseños obtengan las habilidades por nosotros. Ah, pero antes de hacerlo, vuelve a `skills.html.twig`, añade un `{% block title %}`, escribe "Todas las habilidades" y luego `{% endblock %}`:

templates/main/skills.html.twig

```
{% extends nglayouts.layoutTemplate %}

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

Esto, como probablemente sepas, controla el título de la página. Hago esto aquí porque el bloque `title` en realidad no es algo que puedas controlar a través de Maquetaciones. Recuerda: todo lo que construimos en nuestra maquetación pasa a formar parte de un bloque llamado `layout`.

Añadir el diseño de la lista de habilidades

Bien, pulsa "Publicar" en el Diseño de la página de inicio... y crea un nuevo diseño. Utilizaré mi "Diseño 2" favorito y lo llamaré "Diseño de lista de habilidades".

Ya sabes cómo funciona. Empieza por enlazar la zona de cabecera... y la zona de pie de página. A continuación, vamos a crear otro héroe. Añade una columna, dale una clase `hero-wrapper` y pon dentro un bloque "Título" con "Todas las habilidades". Para molar aún más, añade un bloque de texto debajo con algún contenido de introducción.

¡Buen comienzo! Publica la maquetación... para que podamos ir a enlazarla a la página `/skills`. Pulsa "Añadir nueva maquetación" y enlaza esto a la "Maquetación de la lista de habilidades". Luego ve a "Detalles". Esta vez mapearé a través de la Información de la Ruta, establecida en `/skills`. Pulsa Guardar cambios.

Vamos a ver cómo queda nuestro primer intento. Y... ¡no está mal!

Añadir la parrilla de habilidades

Ahora vamos a añadir lo importante. Vuelve al administrador de diseños y edita este diseño.

Debajo de la columna, añade una nueva cuadrícula. Cámbiala de colección manual a colección dinámica. El paquete Contentful nos ofrece dos nuevos "tipos de consulta", o formas de "obtener" datos de Contentful. Utiliza la "Búsqueda Contentful". Es la principal.

Esto te permite elegir qué tipos de contenido mostrar, como todos... o sólo habilidades. Luego podemos ordenarlos, añadir una búsqueda, omitir elementos o limitarlos. Es todo lo que queremos, ¡desde el principio!

¿Qué aspecto tiene? Pulsa "Publicar". Seguro que lo adivinas. ¡Sí! "Funciona"... imprimiendo el título de cada habilidad. Oh, déjame al menos añadir esa clase "contenedor"... para obtener el margen izquierdo y derecho.

Pero, ¡obviamente esto no es lo que queremos! Necesitamos poder darle estilo e imprimir más campos aparte del título. Tenemos el mismo problema en la página de inicio.

Y en realidad, ¡esto es aún más complejo de lo que parece! Cuando personalizamos cómo se muestra una cuadrícula de habilidades, quiero poder hacer que esos elementos se vean de una manera en la página de inicio, y de otra diferente en la página "Habilidades", probablemente más grandes y con más campos impresos.

A continuación: vamos a empezar a aprender el importantísimo tema de cómo podemos anular y personalizar las plantillas de Diseños para que podamos hacer que las cosas se vean exactamente como queremos.

Chapter 19: Temas y sustitución de plantillas

Ahora podemos añadir mucho contenido dinámico a nuestro sitio, como estos bloques estáticos de aquí arriba, cuadrículas o listas. Las cuadrículas y las listas pueden contener elementos de Contentful o de nuestra entidad `Recipe`. Pero para que nuestro sitio brille de verdad, necesitamos flexibilidad sobre el aspecto de estas piezas. Empecemos por lo más sencillo, anulando la plantilla que muestra el aspecto del bloque "Título" para toda nuestra aplicación.

Encontrar plantillas de bloques en el perfilador

Para ello... primero tenemos que averiguar qué plantilla se encarga actualmente de representar este bloque. Una forma fácil de averiguarlo es ir a una página que muestre uno de estos bloques, actualizarla y hacer clic en el icono Twig de la barra de herramientas de depuración web. Abajo, en la parte inferior, vemos todo el árbol. Y si nos fijamos bien, ¡ah ja! ¡Parece que hay una plantilla llamada `block/title.html.twig`!

El propio Layouts también tiene una sección de la barra de herramientas de depuración web realmente bonita. Si vas a "Bloques renderizados", muestra "Definición de bloque: título", "Texto", "Lista" y "Pie". Y, como hemos visto, el Título se renderiza mediante `title.html.twig`.

Hola Temas

Observa que casi todas estas plantillas están dentro de los directorios `themes/standard/`. Layouts tiene un concepto de temas, aunque no necesitaremos crear varios temas a menos que estemos construyendo algún tipo de aplicación multisitio. En nuestro caso, sólo vamos a utilizar el tema incorporado llamado `standard`.

Pero los temas siguen siendo importantes, porque cualquier cosa dentro de un tema puede ser fácilmente anulada colocando una plantilla en el lugar adecuado. Vamos a utilizar esa convención para anular la plantilla Título.

Anular la plantilla Título

¡Vamos a hacerlo! Primero, en el directorio `templates/`, asegúrate de que tienes un subdirectorio `nglayouts/`. Dentro de él, añade uno nuevo llamado `themes/`... seguido de otro subdirectorio llamado `standard/`. Te habrás dado cuenta de que estamos igualando la estructura que hay aquí: `nglayouts/themes/standard/`.

Dentro de esto, como la plantilla de destino se llama `block/title.html.twig`, si creamos esa misma ruta, nuestro `title.html.twig` ganará. Hazlo: añade otro directorio llamado `block/` y un nuevo archivo dentro: `title.html.twig`. Para ver si funciona, escribe un texto ficticio:

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

```
1  OVERRIDING TITLES!!
```

¡Probemos esto! Vuelve a la página Habilidades, actualiza y... no pasa absolutamente nada. Eso es porque la primera vez que creamos este directorio `themes/`, tenemos que borrar la caché.

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

Hazlo... y después, vuelve a probar la página. ¡Yupi! ¡Ahora controlamos cómo se muestra el bloque Título! ¡El poder!

Hacer que la plantilla de título sea más realista

Vale, pero aunque queramos personalizar cómo se muestra el Título... probablemente no queramos empezar de cero. Sería mejor reutilizar parte de la plantilla principal, o al menos utilizarla como referencia.

Pulsa `Shift`+`Shift`, busca `title.html.twig`, y selecciona "Incluir elementos no del proyecto". Abre la del núcleo desde `nglayouts/themes/`.

Vaya. Aquí pasan muchas cosas... incluido el hecho de que amplía otra plantilla: `block.html.twig`. Ábrela.

Contiene muchas funciones básicas, como leer la variable dinámica `css_class`, que contiene las clases CSS que introducimos en el administrador. También gestiona si hay un contenedor o no. ¡Son cosas útiles!

En `title.html.twig`, tiene código para saber si el título es o no un enlace y otras cosas. Podríamos sustituir totalmente esta plantilla e ignorar todo esto si quisiéramos. Pero en lugar de eso, copia la plantilla principal, pégala en nuestra versión:

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

```
1  {% extends '@nglayouts/block/block.html.twig' %}
2
3  {% import '@NetgenLayouts/parts/macros.html.twig' as macros %}
4
5  {% set tag = block.parameter('tag').value|default('h1') %}
6  {% set link = block.parameter('link') %}
7
8  {% block content %}
9      {# Located inside the "content" block to include the context from the parent
   template #}
10     {% set title = macros.inline_template(block.parameter('title').value,
   _context) %}
11
12     <{{ tag }} class="title">
13         {% if block.parameter('use_link').value and not link.empty %}
14             {{ nglayouts_render_parameter(link, {content: title}) }}
15         {% else %}
16             {{ title }}
17         {% endif %}
18     </{{ tag }}>
19 {% endblock %}
```

Y sólo para demostrar que podemos, eliminemos esa clase `title`:

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

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

¡Genial! Ahora ve, actualiza y... vuelve a tener el mismo aspecto que antes. Pero aquí abajo, ¡la clase `title` de `<h1>` ha desaparecido!

Así que la forma más sencilla de controlar el aspecto de algo es encontrar la plantilla que lo renderiza y anularla por completo utilizando esta estructura de directorios `themes/`.

Volvamos a utilizar este truco para personalizar el aspecto de un campo "activo" de Contentful, como este campo de imagen de habilidad. Pero por el camino, vamos a profundizar en algunos conceptos de enorme importancia: las vistas en bloque y los tipos de vista.

Chapter 20: Vistas en bloque y tipos de vista

Vamos a anular completamente otra plantilla. Entra en el Diseño de habilidad individual. Aquí estamos utilizando una entrada de Contentful, que es un "Activo referenciado"... y se está renderizando como esta etiqueta de imagen. ¡Genial!

Bloque "Tipos de vista" / Plantillas

Este es un gran ejemplo de cómo un único tipo de bloque -por ejemplo, el tipo de bloque "Campo de entrada Contentful"- puede tener varios tipos de Vista, lo que básicamente significa "varias plantillas". Cada uno de estos diferentes tipos de Vista será representado por una plantilla diferente. En realidad vemos esto con muchos tipos de bloque diferentes, incluso con el tipo de bloque Rejilla. Añadiré uno aquí abajo temporalmente. Tiene un tipo de Vista que te permite cambiar entre Lista y Cuadrícula. Sí, los bloques Lista y Cuadrícula son en realidad el mismo tipo de bloque internamente: sólo tienen un tipo de vista diferente, lo que significa que cada uno es representado por una plantilla distinta. Adelante, borra eso.

En cualquier caso, cada tipo de bloque puede tener uno o más tipos de vista. Y en realidad quiero profundizar un poco más en este concepto de "vistas". Busca tu terminal y ejecuta:

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

Estoy depurando la configuración que podría vivir bajo la tecla `view` debajo de la tecla `netgen_layouts`:

config/packages/netgen_layouts.yaml

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

Cuando ejecutes esto, verás un montón de configuraciones. Observa que hay varias claves raíz, como `parameter_view`, `layout_view`, y algunas otras. Pero en realidad sólo hay dos

que nos interesen: `block_view`, de la que hablaremos ahora, y `item_view`, que controla cómo se representan los elementos de una Lista o Cuadrícula. En realidad, ya vimos esto antes cuando personalizamos cómo se representaba nuestro "elemento" Receta dentro de una Lista o Cuadrícula. Pronto hablaremos más de ello.

La configuración de la vista en bloque

De todos modos, para ampliar las vistas de bloque, ejecuta el mismo comando, pero añade `.block_view`

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

Las vistas de bloque, en pocas palabras, controlan cómo se muestran los tipos de bloque completos. Por ejemplo, podemos ver cómo se visualiza el "Bloque de título"... o el "Bloque de texto", o cómo se visualiza el "Bloque de lista".

Esta configuración `block_view` puede tener varias claves debajo, como `default`, `app`, y `ajax`. Y sabemos lo que significan. `default` significa que se utilizan en el frontend, `app` significa que se utilizan en la sección de administración y `ajax`, que no es tan común, se utiliza en el frontend para las llamadas AJAX. Así que para anular la plantilla del frontend para un bloque, en realidad queremos decir que queremos anular su "vista" de bloque bajo la clave `default`.

Vamos... a ampliarlo una vez más añadiendo `.default`:

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

La configuración "coincidente

Estas son todas las vistas de bloque que se utilizarán en el frontend. Lo más complicado de éstas es la parte `match`.

Cuando defines una "vista de bloque", es bastante habitual definir la plantilla que debe utilizarse cuando dos cosas coinciden. Busca "list\grid": es un gran ejemplo. Tiene dos elementos `match`: `block\definition` se establece en `list` porque, técnicamente, el "Tipo de bloque" para los bloques Lista y Cuadrícula se llama `list`. La segunda condición de coincidencia es `block\view_type` establecido en `grid`.

Juntas significan que si se está renderizando un bloque cuyo `block\definition` es `list` y cuyo `block\view_type` es `grid`, utiliza esto.

Por cierto, ambas cosas pueden verse muy claramente desde la barra de herramientas de depuración web. Ve a la página de inicio, haz clic en la barra de herramientas de depuración web Diseños y ve a "Bloques renderizados". Aquí abajo... ¡mira esto! Puedes ver "Definición de bloque: ¡Lista", "Tipo de vista: cuadrícula"! Y luego apunta a la plantilla que se ha renderizado. En este caso, se refiere a esta cuadrícula de aquí.

Entonces... ¿por qué el bloque Título es renderizado por `title.html.twig`? Podemos verlo en la configuración. Busca "título"... aquí lo tenemos. Esto dice: si el `block\definition` es `title` y el `block\view_type` es `title`, utiliza esta plantilla. Este es un ejemplo de un tipo de Bloque que sólo tiene un tipo de Vista. Así que, en la práctica, ésta es la vista que se utiliza para todos los bloques de título.

Buscar y anular la vista de activos de campo de Contentful

Bien, recordemos nuestro objetivo original: anular la plantilla que renderiza esta imagen. Sabemos que se trata de un "Campo de entrada Contentful" y que tiene un tipo de Vista "Activos referenciados". Así que... ¡podemos encontrarlo aquí!

Busca "activos" y... ¡ahí está! Así que si `block\definition` es `contentful_entry_field` y `block\view_type` es `assets`, ¡ésta es la plantilla! Esto significa que si queremos anular sólo el tipo de Vista `assets` de la entrada Contentful, ésa es la plantilla que tenemos que anular.

Y sí, podríamos haberlo encontrado muy fácilmente yendo a la barra de herramientas de depuración web y encontrando allí la plantilla. Pero ahora entendemos un poco mejor cómo se representan los bloques y cómo cada bloque puede tener varias vistas para que podamos elegir cómo se representan. Más adelante, añadiremos un "tipo de vista" adicional a un bloque existente.

Bien, manos a la obra. La ruta comienza con la normal `nglayouts/themes/standard/`, luego necesitamos `block/`, seguida de esta ruta. Así que dentro de nuestro directorio `block/`, crea un nuevo subdirectorio llamado `contentful_entry_field/`. Y dentro de éste, un nuevo `assets.html.twig`. Por ahora, sólo diré `ASSET`:

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

```
1 ASSET
```

Vale Gira hacia el frontend y... ¡sí! ¡Lo ve al instante! ¡Ya tenemos el control!

Hacer la plantilla más elegante

Como antes, probablemente no queramos reemplazar toda la plantilla. En lugar de eso, abre la plantilla principal - `assets.html.twig` - para que podamos robarla, tomarla prestada. Temporalmente, cópiala entera, pégala:

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 %}
```

Y... ¡sí! Funciona.

Contentful es bastante avanzado... y puedes ver que admite campos que contienen una sola imagen, así como múltiples imágenes. Puedes mantener esto tan flexible como quieras, pero también puedes hacerlo a tu gusto. Voy a simplificar drásticamente esta plantilla... y a sustituirla por una imagen muy sencilla. Para el `src`, pegaré algo de código:

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

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

{% block content %}
    {% set field = block.dynamicParameter('field') %}
    {{ dump() }}
    <img src="{{ field.value.file.url }}?h={{ block.parameter('height').value }}&w={{ block.parameter('width').value }}" />
{% endblock %}
```

Todas las partes Twig de este código estaban antes en la plantilla. Esto también muestra un superpoder de Contentful en el que puedes controlar el tamaño de la imagen. Llamar a `block.parameter()` nos permite leer las opciones del administrador de diseños, donde antes configuramos este bloque para que tuviera una anchura y una altura de 200.

¡Veamos qué aspecto tiene! Actualiza. ¡Sí! ¡Parece que ha funcionado!

Elegir si mostrar o no opciones complejas

Pero quiero hacer una pequeña advertencia sobre la personalización de plantillas: asegúrate de no perder la flexibilidad que necesitas. Por ejemplo, sabemos que podemos añadir clases CSS adicionales a cualquier bloque a través del admin.

Si lo hiciéramos ahora, no funcionaría porque... ¡simplemente no renderizaríamos esas clases! Y, eso podría estar bien. Pero si quieres admitirlas, tendrás que asegurarte de añadirlas. En este caso podemos decir `class="{{ css_class }}"`, que es una de las variables que vimos antes. Y ya que estamos aquí, añadamos también un atributo `alt` establecido en `field.value.title`:

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

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

Cuando probemos esto... ¡Me encanta! Ahí está el atributo `alt` y ahí está nuestra clase, incluidas algunas clases principales que Layouts siempre añade a esa variable.

Vale, acabamos de hablar de las vistas de bloque: cómo se configuran las plantillas para bloques enteros. A continuación, vamos a hablar de las vistas de elementos: cómo personalizamos la plantilla que se utiliza al mostrar un elemento dentro de una cuadrícula o lista. Utilizaremos esto para dar estilo a nuestros elementos de habilidad.

Chapter 21: Inmersión profunda en las vistas de elementos

Cuando se trata de personalización, puedes hacer mucho daño mirando qué plantillas se están renderizando y utilizando el sistema de temas para anularlas. Pero hay algunos casos en los que necesitarás ser aún más específico.

Por ejemplo, supongamos que queremos modificar la plantilla "elemento" para ver cómo se muestra la cuadrícula de habilidades en la página de inicio. Si compruebas aquí la barra de herramientas de depuración web y te desplazas hacia abajo... buscaré "contentful"... ah, ya está. Puedes ver `grid.html.twig`... que renderiza `item/contentful_entry.html.twig`. Para personalizar el elemento, podríamos anular esa plantilla. Muy fácil.

El problema es que, en Contentful, tenemos varios tipos de contenido: tenemos Habilidades y Anuncios. Así que si modificamos esta plantilla, la modificaremos tanto para Habilidades como para Anuncios... y probablemente queramos que tengan un aspecto diferente. Entonces, ¿cómo podemos resolver esto?

Profundizando en item_view Config

Antes hemos ejecutado `debug:config netgen_layouts view` y hemos hablado de las dos secciones principales que hay aquí debajo: `block_view` (que controla cómo se muestran los bloques) y `item_view`.

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

Como ya he dicho varias veces, algunos bloques, como Grid y List, muestran elementos individuales. En esta configuración de `item_view` es donde definimos esas plantillas. Y verás algunas claves raíz familiares: `default` para el frontend, `ajax` para las llamadas AJAX y `app` para el admin. Una vez más, esto utiliza la configuración `match` y... ¡eh! ¡Vemos nuestra

entrada aquí! ¿Recuerdas `recipes_default`? Lo configuramos dentro de nuestro archivo de configuración, y es una de las dos plantillas de elementos reales que tenemos ahora mismo:

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
```

Hay una para recetas, y luego Contentful tiene una para todos los elementos de Contentful.

Pero nuestro objetivo es anular esta plantilla sólo cuando el elemento sea una habilidad, como en este caso. ¿Y cómo lo hacemos? Añadiendo nuestro propio `item_view` a esta lista que coincide con ese único tipo de contenido. ¡Vamos a hacerlo!

Añadir un item_view personalizado

Por aquí... estamos bajo `item_view`, `default` para el frontend y tenemos la única entrada de antes: `recipes_default`. Vamos a añadir otra. Llámala `contentful_entry/skill`, aunque esta clave en concreto no supone ninguna diferencia. Debajo de ella, pon `template` en `@nglayouts/item/contentful_entry`, seguida de `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
```

Antes, utilizábamos `nglayouts` sin el `@`... sólo porque te dije que `nglayouts/` era un buen directorio para organizar cosas. Internamente, Layouts utiliza `@nglayouts` en las rutas de sus plantillas. ¿Cuál es la diferencia? Al añadir el `@`, nos estamos enganchando al sistema de temas. Así, como tenemos un directorio `templates/nglayouts/` con `themes/standard/` dentro, utilizará nuestra plantilla. Puedes utilizar `@nglayouts` o simplemente `nglayouts`. Sólo quería que entendieras la diferencia porque verás la sintaxis `@nglayouts` por todas partes.

Coincidencia con un solo tipo de contenido

La clave realmente importante aquí es `match`. Queremos que coincida sólo cuando estemos trabajando con un `contentful_entry`. Vale, copia `match` de la config... y pega.

Pero tenemos que ser más específicos. También necesitamos coincidir sólo cuando el tipo de contenido sea una habilidad. Pero, ¿cómo lo hacemos? ¿Qué comparadores hay disponibles? Hay una lista básica... pero, ¿ha añadido Contentful algún comparador adicional que podamos aprovechar?

He aquí un pequeño truco para ver la verdadera lista de elementos de `match`. Es un poco técnico, pero funciona de maravilla. Ejecuta:

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

¿Qué hace esto? Bueno, cualquiera puede crear un comparador personalizado, como `foo\bar`. Para ello, creas una clase y le das esta etiqueta. Buscando todos los servicios con esa etiqueta, podemos encontrar todos los matchers existentes en el sistema.

Y... ¡mira qué lista! Aquí hay uno interesante: `contentful\content_type`. Seguro que podemos utilizarlo. Inténtalo: `contentful\content_type` ajustado a `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
```

Bien, vamos a crear la plantilla. Dentro de `themes/standard/`, en lugar de `block/`, esta vez, crea un directorio llamado `item/`... luego `contentful_entry/`, y luego `skill.html.twig`. De momento pon un texto ficticio:

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

```
CONTENTFUL SKILL!
```

Vale, si esto funciona, cuando actualicemos, estos elementos -que son habilidades de Contentful- deberían volver a renderizarse utilizando nuestra nueva plantilla. Pero cuando lo intentamos... no cambia absolutamente nada. ¿Qué ha ocurrido?

¡Orden de configuración incorrecto!

Vuelve a tu terminal y ejecuta

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

de nuevo. Todo parece correcto... excepto el orden. Esta de Contentful está al principio de la lista... y las nuestras nuevas están al final. ¿Y adivina qué? Cuando Layouts intenta averiguar

qué plantilla debe renderizar, lee la lista de arriba a abajo y encuentra la primera que coincide: exactamente como funciona el sistema de enrutamiento de Symfony.

Así, primero mira `contentful_entry`, ve que la `value_type` es `contentful_entry`... y la utiliza. Nunca llega al `contentful_entry/skill` de la parte inferior.

Para solucionarlo, vamos a utilizar un elegante truco de configuración de prefijo. Hagámoslo a continuación.

Chapter 22: Configuración previa

Estoy bastante seguro de que nuestro nuevo `item_view` está configurado correctamente. Tenemos `item\value_type: contentful_entry`, que sé que es correcto... y luego estamos utilizando `contentful\content_type` ajustado a `skill` para que esto sólo afecte a las habilidades:

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
```

Pero... parece que no funciona en el frontend. Antes, cuando ejecutamos `debug:config`, vimos que el problema reside en el orden de la configuración. Layouts lee de arriba abajo cuando decide qué "vista" utilizar. Así que mira ésta primero, ve que la `value_type` es `contentful_entry`... y simplemente se detiene. Para solucionarlo, tenemos que invertir nuestra configuración.

Vale, entonces... ¿por qué está en este orden para empezar? ¿Por qué nuestra configuración aparece al final? Esto se debe a la forma en que Symfony carga la configuración: primero carga la configuración del bundle - como la del paquete Contentful o Layouts - y luego carga nuestros archivos de configuración. Y ese suele ser el orden que queremos Nos permite anular la configuración establecida en los bundles.

Pero en este caso, queremos lo contrario. ¿Cómo lo conseguimos? Pidiendo a Symfony que preañada nuestra configuración.

Configurar el Prepend

En el directorio `config/`, crea un nuevo directorio llamado `prepends/` y mueve la configuración de Netgen Layouts a él. Esto evitará que Symfony cargue ese archivo de la forma normal: vamos a cargarlo manualmente.

El siguiente paso es un poco técnico. En `src/`, crea una clase "extensión" llamada, qué tal, `AppExtension`. Voy a pegar el código: puedes cogerlo del bloque de código de esta página:

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));
    }
}
```

Esto carga nuestro archivo de configuración de forma normal... excepto que se le añadirá una extensión.

Paso final. Para llamar a este método, abre la clase `Kernel`. Después de `use MicroKernelTrait`, añade `configureContainer as baseConfigureContainer`:

src/Kernel.php

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

Esto añade el método `configureContainer` de `MicroKernelTrait` a esta clase como haría normalmente un trait... excepto que lo renombra a `baseConfigureContainer`. Hacemos esto para poder definir nuestro propio método `configureContainer()`. Copia la firma `configureContainer()` del trait, pégala, pulsa "OK" para añadir las sentencias `use` y luego llama a `$this->baseConfigureContainer()` pasando por `$container`, `$loader` y `$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
    }
}
```

El método `configureContainer()` del trait se encarga de cargar `services.yaml` y todos los archivos de `config/packages/`. Todo eso son cosas buenas que queremos seguir haciendo. Pero después de hacer eso, añade una cosa más: `$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());
    }
}
```

De nuevo, sí, esto es fastidiosamente técnico. Pero gracias a estas dos piezas, nuestra configuración `netgen_layouts.yaml` estará preconfigurada.

¡Compruébalo! Vuelve a ejecutar el comando `debug:config`:

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

Desplázate hacia arriba y... ¡sí! ¡Nuestra configuración está ahora arriba! Y cuando actualizamos... ¡woohoo! ¡Vemos el texto!

A continuación: vamos a hacer que esta plantilla se muestre exactamente igual que las habilidades codificadas. A continuación, crearemos una segunda plantilla de elementos para personalizar la representación del tipo de contenido "Anuncio" de Contentful.

Chapter 23: Plantilla de elementos Contentful

¡Nuestra plantilla "ítem" para habilidades ya se está utilizando! Así que, ¡vamos a terminarla!

Ya sabemos qué aspecto queremos que tengan las habilidades... así que vamos a robarlo de `templates/main/homepage.html.twig`. Busca el bloque `featured_skills`, copia el aspecto de una de esas habilidades y pégalo en `skill.html.twig`. Añadamos también `dump(item.object)` en la parte superior. Ya hemos creado antes una plantilla de artículo, así que sabemos que `item.object` debería darnos el "objeto" subyacente que representa esta entrada de Contentful.

Si nos dirigimos y actualizamos... ¡genial! Esto vuelca un objeto `ContentfulEntry`. Y, aunque no puedas verlo desde aquí, esta clase tiene un método `get()` que podemos utilizar para leer cualquiera de los datos subyacentes de Contentful.

Para las habilidades, si escarbamos un poco... tenemos campos como `title` y `shortDescription`. ¡Usémoslos! Por ejemplo, en el `<h3>`, digamos `{{ item.object.get('title') }}`. Y... ¡sí! Eso renderiza el título.

Para el `<img src="">`, sustituye lo de `asset()` por `item.object.get('image')`, seguido de `.file.url`, que es específico de Contentful. Rellena también el atributo `alt`con `item.object.get('title')`.

Lo último que tenemos que actualizar es la URL. Pero Si hubiéramos creado una página de "demostración de habilidades" en Symfony, ¡podríamos utilizar la función de ruta de Twig y enlazar a esa ruta! Sin embargo, cada página de habilidad se crea en realidad a través de una ruta dinámica gracias al bundle Contentful. Y, para crear esas rutas, utiliza el sistema de rutas CMF.

Así que, para enlazar, tenemos que utilizar ese sistema. Por ejemplo, `path('cmf_routing_object')` y pasar `_route_object` a `item.object`.

Si estuvieras utilizando Sylius o Ibexa CMS, utilizarías alguna función de su sistema para crear este enlace: esto es específico del sistema de enrutamiento CMF.

Si nos dirigimos a él y lo probamos... ¡sí! Y si hacemos clic en él... ¡doble sí!

Celebrémoslo eliminando el `dump()`... y borrando el bloque `featured_skills` de nuestra plantilla de página de inicio. Ya no lo necesitaremos. Incluso podemos rehacer este `<h2>` dentro del administrador de diseños. Hagámoslo: añade un bloque Título llamado "Habilidades destacadas", hazlo "Título 2"... y dale la misma clase CSS: `text-center mb-4`.

La Rejilla ya está en un contenedor... pero queremos todo esto en un contenedor. Así que añade una Columna, envuélvela en un Contenedor, mueve los bloques Rejilla y Título dentro de ella... entonces ya no necesitaremos un Contenedor justo ahí. Elimina el bloque "Características Habilidades"... y finalmente pulsa "Publicar y continuar editando". Mientras esperamos, elimina también ese bloque de la plantilla Twig.

Y ahora... ¡sí! ¡El aspecto es perfecto!

La vista de elementos publicitarios

Vale, ya que estamos hablando de vistas de elementos, vamos a personalizar la plantilla de elementos para nuestro otro modelo de contenido dentro de Contentful: Publicidad. Sólo vamos a renderizarlo en un lugar, en una página específica de habilidades justo... aquí. Vamos a comprobarlo.

Publica este diseño... y luego edita el diseño individual de la habilidad. Antes hemos utilizado el bloque Campo de entrada Contentful para mostrar el campo `advertisements`, que es una "entidad referenciada". Sí, si modificas una habilidad en Contentful, abajo en la parte inferior, el campo "Anuncio" te permite elegir entre los Anuncios de nuestro sistema.

Haz clic en el icono Twig de la barra de herramientas de depuración web... busca "elemento", y desplázate hacia abajo.. No es ninguna sorpresa: está utilizando la plantilla estándar "item" de Contentful. Y, buena noticia, ya sabemos cómo anularla.

Ve a nuestra configuración, copia la sección `contentful_entry/skill` y pégala a continuación. A continuación, sustituye `skill` por `ad` para el nombre de la sección, `template` y, por último, establece `content_type` en `advertisement`... porque ése es el nombre interno de ese tipo en Contentful.

¡Vale! Vamos a añadir esa plantilla. En `contentful_entry`, crea un nuevo archivo llamado `ad.html.twig`... y luego simplemente añade algo de texto: `Advertisement`.

Momento de la verdad. Vuelve, actualiza... ¡ya lo tenemos! ¡Ha sido fácil!

Para el contenido real de la plantilla, simplemente pegaré algo. Una vez más, utilizaremos `item.object.get()` para leer el campo `url`. También hay un campo `image` y un campo `shortText`. Y ahora... ¡ya lo tenemos!

Lo siguiente: ¿Qué pasaría si quisiéramos crear una Cuadrícula de elementos en nuestro sitio, pero hacer que esa única Cuadrícula tuviera un aspecto diferente al de todas las demás cuadrículas del sitio? Podemos hacerlo creando una "vista de bloque" adicional para un bloque existente.

Chapter 24: Vistas de bloques y definiciones de bloques

Vamos a crear un diseño para nuestra página de "receta individual" para poder personalizarla un poco más. Me encanta que podamos crear nuevos diseños sobre la marcha, siempre que haya que modificar una página.

Añadir y asignar un nuevo diseño

Añade una nueva maquetación, elige nuestra Maquetación 2 favorita y llámala "Maquetación de receta individual". A estas alturas, ya conoces el procedimiento. Empieza por vincular la zona del Encabezado... y luego la zona del Pie.

¡Genial! Y luego, como vamos a aplicar esto a una página normal que ya existe en Twig, añade un bloque "Vista completa", que renderizará el bloque `body` de nuestra plantilla.

Un comienzo sólido. Pulsa "Publicar"... para que podamos mapear esto. Añade un nuevo mapeo, vincúlalo a nuestro "Diseño de receta individual"... y pulsa "Detalles". esta vez, vamos a vincularlo a través del nombre de la ruta. Para el nombre de la ruta, abre `src/Controller/RecipeController.php`. Aquí está: `app_recipes_show`. Pégalo, pulsa "Guardar cambios" y... ¡a probar!

Aún no deberíamos ver ninguna diferencia y... no la vemos. Pero podemos ver que está utilizando nuestro diseño

Bien, ¡vamos a animar un poco esta página! Vuelve al administrador de diseños y edita el "Diseño de receta individual". Añade una nueva Rejilla y cámbiala por una "Colección dinámica"... que utilice "Contentful Search". Como hicimos antes, carga Habilidades, muestra la más reciente primero y limita a 3.

Vale, si "Publicamos y seguimos editando"... y luego actualizamos... ¡guau! Es genial que ahora podamos ponerlas en cualquier sitio. Aunque, envolvamos eso y contenedor. Y... ya está.

Hasta ahora, todo esto es fácil. ¿Listo para la complicación? Quiero personalizar el aspecto de esta cuadrícula: Quiero tener una receta grande a la izquierda y luego dos recetas más pequeñas a la derecha. Pero no quiero cambiar el aspecto de la cuadrícula en otras partes de nuestro sitio, como en la página de inicio. Así que la pregunta es cómo: ¿podemos cambiar el aspecto de esta cuadrícula sólo en esta página?

Los tipos de vista cuadrícula/lista

Haz clic en la cuadrícula y ve a la pestaña de diseño. Resulta que una cuadrícula es en realidad un bloque "Lista" que tiene dos "tipos de vista": lista y cuadrícula.

Dirígete a tu terminal y ejecuta:

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

Oh, pero escribe `netgen` correctamente. Esto muestra la configuración de cómo se representan los bloques. Busca la sección `default`... y desplázate un poco hacia abajo. Aquí: vemos los dos tipos de vista para los bloques de lista y de cuadrícula. Como ya he mencionado, resulta que en realidad ambos forman parte del mismo tipo de bloque llamado `list`. Sólo son dos tipos de vista diferentes: uno llamado `list` y otro `grid`. Cuando cambias el "tipo de vista" en el administrador de diseños, en realidad estás cambiando la plantilla que se utiliza para mostrar ese bloque.

Definiciones de bloque

Ejecuta ese mismo comando, pero en lugar de `view.block_views`, marca `block_definitions`:

```
php bin/console debug:config netgen_layouts block_definitions
```

Las definiciones de bloque es donde defines lo que son realmente los bloques. Así, cada clave raíz de esta configuración representa un bloque diferente que podemos utilizar en el área de administración. Busca el que se llama `list`: aquí está. Define cosas como qué campos de

formulario se muestran en el área de administración y qué "tipos de vista" tiene. Tiene dos: lista y cuadrícula. Layouts lee esta configuración para mostrar el elemento `select` de tipos de vista en el área de administración. Luego, una vez que seleccionamos el tipo de vista, utiliza la configuración de `block_views` para saber qué plantilla debe mostrar.

Vale, basta de configuraciones profundas y teoría. Vamos a darnos una nueva forma de representar listas creando un nuevo tipo de vista. Eso a continuación.

Chapter 25: Vista de bloque personalizada

Éste es el plan. Vamos a añadir un nuevo "tipo de vista" a la definición del bloque de lista. Luego vamos a asignarlo a una plantilla a través de `block_views`.

Actualizar la "definición de bloque

Para el paso 1, abre nuestro archivo `netgen_layouts.yaml` y, en cualquier lugar, añade `block_definitions`. Esta configuración puede utilizarse para crear bloques totalmente nuevos o para cambiar opciones de bloques existentes, que es lo que queremos. Para ello, tenemos que repetir la configuración aquí: `list` & `view_types`. Así, `list` `view_types` y luego añadir el nuevo. Llamémoslo `one_by_two` -esa clave puede ser cualquier cosa- y démosle un nombre: `1x2 Featured Grid`:

config/prepends/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 12
13     block_definitions:
14         list:
15             view_types:
16                 one_by_two:
17                     name: 1x2 Featured Grid
↕  // ... lines 18 - 52
```

Sólo con eso, si vamos y actualizamos el área de administración... y hacemos clic abajo en la cuadrícula, ¡tenemos un nuevo tipo de vista! Si cambiamos a ella... no aparece nada en el área de administración. Y si pulsamos "Publicar y continuar editando"... en el frontend... tampoco se muestra nada. ¡Sí!

Haz clic en el enlace Diseños de la barra de herramientas web y... cerca de la parte inferior, ah. Se está mostrando `invalid_block.html.twig`. La definición del bloque es `list` y el tipo de vista es `1x2 Featured Grid`. El problema es que aún no hemos definido una "vista de bloque" para esa combinación. Así que vuelve a "bloque no válido".

Añadir la vista de bloque Admin

Vale, en `view`, ya hemos creado varias "vistas de elemento". Ahora añade `block_view` para que podamos crear la primera de ellas. Vamos a registrar tanto una vista de administrador como una vista de frontend. Porque... en el área de administración, actualmente no se muestra nada. Añade `app` para el admin y la siguiente clave no importa. Para la plantilla, como la vista admin no es demasiado importante, vamos a reutilizar la plantilla "grid" del núcleo admin, que puedes encontrar mediante el comando `debug:config`.
Es `@NetgenLayoutsStandard/app/block/list/grid.html.twig`.

Ahora añade `match`. Queremos utilizar esta plantilla si `block\definition` es `list` y `block\view_type` es `one_by_two`... asegurándonos de que coincide con la clave que hemos utilizado antes en la definición del bloque:

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
```

¿Cómo sabía que debía utilizar `block\definition` y `block\view_type`? ¡Utilizando nuestro comando favorito `debug:config`! Siempre es una buena guía a seguir.

En cualquier caso, esto debería arreglar el área de administración. Y... ¡lo hace!

Vista en bloque del frontend

Para la vista del frontend, duplica toda esa sección... pero utiliza `default`. Esta clave está bien, no importa, y cambia la plantilla a, qué tal, `@nglayouts/block/list/one_by_two_list.html.twig`. La sección de coincidencia ya es perfecta:

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
```

Vale, ¡vamos a hacer esa plantilla! Ya tenemos `templates/nglayouts/themes/standard/block/`... así que, crea el nuevo subdirectorio `list` y luego el archivo: `one_by_two_list.html.twig`. Empieza diciendo `1x2`:

templates/nglayouts/themes/standard/block/list/one_by_two_list.html.twig

```
1x2
```

¡Vamos a comprobarlo! En el frontend, actualiza y... ¡ahí está nuestro pequeño 1x2!

Personalizar la plantilla del frontend

¡Vamos a darle vida! Como esto representa un bloque "lista", nuestra plantilla probablemente tenga acceso a alguna variable que represente los "elementos". Para hacer trampas, lo que siempre es una buena opción para los desarrolladores, echemos un vistazo a la plantilla de rejilla principal: `grid.html.twig` del directorio `themes/`.

¡Vaya! Como muchas plantillas de núcleo, ¡aquí hay un montón de cosas! Puedes elegir lo que quieres conservar o eliminar. Lo más importante es esta variable `collection_html`: hacen un bucle sobre `collections[collection_identifier]`... donde `collection_identifier` es en realidad sólo la palabra `default`. Así que se repite `collections.default`. Luego incluye una plantilla. Esa variable `templateName` se establecerá en algo como `grid/` el número de columnas `.html.twig`. Por ejemplo, si la cuadrícula está configurada para utilizar 3 columnas, utilizaría `3_columns.html.twig`. Esa plantilla añade el `div` necesario para cada columna en una configuración de 3 columnas... y luego llama a `nglayouts_render_result()`. Eso renderiza el "elemento".

De todos modos, si alejas el zoom, la plantilla básicamente hace un bucle sobre la variable `collections` y llama a `nglayouts_render_result()` en cada una de ellas.

De vuelta a nuestra plantilla, voy a pegar un código que hace algo parecido:

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 %}
```

Sí, ampliamos `block.html.twig`, igual que hace la plantilla principal, luego hacemos un bucle sobre `collections.default`, añadimos un `div` y mostramos cada elemento. Así que esto es efectivamente una versión más simple de lo que hace una cuadrícula.

¿Y qué aspecto tiene? Actualiza y... ¡sí! ¡Parece una cuadrícula!

Pero recuerda el objetivo: una gran habilidad a la izquierda con dos habilidades más pequeñas a la derecha. Para conseguirlo, pegaré la versión 2 de mi plantilla. Aquí no hay nada especial. En lugar de hacer un bucle, esto renderiza la tecla 0, luego las teclas 1 y 2:

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 %}
```

Y ahora... ¡sí! ¡Es exactamente lo que quería!

Sin embargo, te haré la misma advertencia que te hice antes cuando modificamos las plantillas de los "elementos" principales. No estamos incluyendo todas las cosas personalizadas que hay en la plantilla principal. Si necesitas soportar una opción personalizada, asegúrate de incluir ese código.

Ocultar opciones de bloque para un tipo de vista de bloque

Y en realidad, una cosa de aquí -el número de columnas- no es algo que necesitemos. Es algo que podemos configurar para el bloque... pero no es relevante en absoluto cuando utilizamos nuestro nuevo tipo de vista.

¿Podríamos... ocultar esa opción al utilizar nuestro tipo de vista? ¡Sí! Vuelve a tu terminal y depura de nuevo la configuración de `block_definitions`:

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

Busca `one_by_two`. Podríamos configurar esta clave `valid_parameters` para eliminar una opción del bloque. El tipo de vista `list` hace exactamente eso. No lo haré, pero así es como se hace.

Vale, vuelve al sitio y ve a la página "Todas las habilidades". Sí... las cosas siguen sin estar bien. En este diseño, utilizamos una cuadrícula para representar los elementos. Y esa cuadrícula se ve bien en otras páginas, pero no aquí, donde se supone que las habilidades son el contenido principal de la página. A continuación, vamos a aprender cómo podemos personalizar la representación de estos elementos sólo para esta cuadrícula.

Chapter 26: Tipo de vista de elemento personalizado

La cuadrícula de habilidades de la página `/skills` tiene un aspecto terrible. Vamos a buscar el diseño para eso: Diseño de lista de habilidades. Vale, esto es una cuadrícula normal... y se muestra como cualquier otra cuadrícula del sitio. Quiero personalizarlo, pero no quiero que el propio bloque de la cuadrícula se muestre de forma diferente: tenerlo en mosaico así está muy bien. Lo que realmente quiero es cambiar cómo se muestra cada elemento dentro de la cuadrícula... pero sólo en esta situación. ¿Cómo podemos hacerlo?

Hola "Tipos de vista de elementos"

Ve a tu terminal y ejecuta nuestro comando favorito `debug:config`, esta vez en `block_definitions`:

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

Esta es, como hemos aprendido, la configuración de todos los bloques de nuestro sistema. Y ¡mira esto! Una parte de la configuración de la que aún no hemos hablado es `item_view_types`. Para cada "tipo de vista de bloque", como `one_by_two`, `list`, o `grid`, existe también `item_view_types`. Hasta ahora, todas ellas tienen una sola llamada `standard`.

No es muy común, pero para un determinado tipo de vista -como `one_by_two` o `list` - puedes especificar múltiples formas de representar los elementos dentro de ese tipo de vista. Éstas se denominan `item_view_types`. `Standard` es la predeterminada, y significa que los elementos se mostrarán de la forma "normal".

Así que éste es nuestro objetivo: para el tipo de vista `grid` existente, vamos a añadir un nuevo "tipo de vista de elementos". A grandes rasgos, esto nos permitirá, al configurar una cuadrícula, elegir una forma diferente de mostrar los elementos.

Para empezar, en nuestra configuración, busca `block_definitions`. Actualmente tenemos `list`, `view_types`, y `one_by_two`. Ahora añade `grid` para que podamos anular ese tipo de vista existente. Añade `item_view_types` con uno nuevo llamado, qué tal, `skill_big_view`. Verás cómo utilizamos esa clave en un segundo. Dale también un nombre legible por humanos:

config/prepends/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 12
13     block_definitions:
14         list:
15             view_types:
↕  // ... lines 16 - 17
18                 grid:
19                     item_view_types:
20                         skill_big_view:
21                             name: Skills Big View
↕  // ... lines 22 - 77
```

¿Qué ha hecho eso? Actualiza el área de administración... haz clic abajo en la Cuadrícula... y asegúrate de que estás en la pestaña "Diseño". Tenemos un nuevo "Tipo de vista de artículo" seleccionado! Aparece "Estándar", que es la predeterminada, y luego nuestra nueva "Gran vista de habilidades"!

Selecciónala y pulsa "Publicar y continuar editando". ¿Qué cambiará esto en el frontend cuando actualicemos? Absolutamente nada Eso es porque ahora necesitamos una nueva "vista de elemento" que coincida con esto.

Añadiendo la "Vista de elemento" para el nuevo "Tipo de vista de elemento".

De vuelta en nuestra configuración, desplázate hasta `item_views`. Debajo de `default`, copia la sección `contentful_entry/skill` y pégala arriba.

Lo ponemos arriba porque el orden es importante: necesitamos que esta nueva vista de elemento pueda coincidir antes que la otra. Observa. Llama a esto `contentful_entry/skill_big_view` y cambia la plantilla a `@nglayouts/item/contentful_entry/skill_big_view.html.twig`. Seguimos queriendo que coincida cuando `item\value_type` sea `contentful_entry` y

`contentful\content_type` sea `skill`... pero sólo si el emparejador llamado `item\view_type` es igual a la clave que creamos antes `skill_big_view`:

config/prepends/netgen_layouts.yaml

```
1  netgen_layouts:
↕  // ... lines 2 - 22
23     view:
24         item_view:
↕  // ... lines 25 - 31
32             # default = frontend
33             default:
↕  // ... lines 34 - 38
39                 contentful_entry/skill_big_view:
40                     template:
   '@nglayouts/item/contentful_entry/skill_big_view.html.twig'
41                     match:
42                         item\value_type: 'contentful_entry'
43                         contentful\content_type: 'skill'
44                         item\view_type: 'skill_big_view'
↕  // ... lines 45 - 77
```

Gracias a esto, si el usuario selecciona este como su "Tipo de vista de elemento" para una cuadrícula de habilidades, entonces los tres coincidirán. Pero si el usuario elige el tipo de vista de elemento por defecto `Standard`, no coincidiría con esto... pero sí con lo de abajo.

Vamos a añadir la plantilla. Dentro de `item/contentful_entry/`, crea el nuevo archivo: `skill_big_view.html.twig`. Dentro, digamos `BIG VIEW`:

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

```
1  BIG VIEW
```

¡Vamos a probarlo! Asegúrate de que la plantilla está publicada... y luego en el frontend... ¡ya lo tenemos! ¡El resto es fácil! Como ya hemos creado varias plantillas de vistas de artículos... Me limitaré a pegar el resto. Aquí no hay nada nuevo.

Pero ahora... ¡sí! Este es el aspecto que queremos.

Cambiar la "vista de artículo" artículo por artículo

Por cierto, ahora que nuestra vista de bloque "Rejilla" tiene múltiples "tipos de vista de elemento" -esa es nuestra configuración aquí arriba- tenemos el poder, elemento por elemento, de controlarlo. ¿Ves este "Anular tipo de vista de ranura"? Esto básicamente dice;

> *“¡Yo Layouts! Quiero cambiar el primer elemento de esta lista para que utilice el tipo de vista "Estándar".”*

Pulsaré "Publicar y continuar editando" y ahora... ¡puedes ver que sólo el primer elemento utiliza el tipo de vista Estándar! Eso... obviamente no es lo que queremos en nuestro sitio, así que volveré atrás y utilizaré "Sin anulaciones". Pero es un concepto muy potente.

Y... ¡guau! ¡Sólo queda un capítulo! Un problema común con los Diseños es trabajar con el espaciado vertical: simplemente asegurarnos de que el espaciado es correcto entre todos nuestros componentes. Podríamos controlarlo añadiendo clases CSS a los bloques individuales. Pero, ¿no estaría bien que todos los bloques del sistema tuvieran un bonito desplegable en el que pudiéramos seleccionar los márgenes superior e inferior automáticamente? ¿Cómo podemos modificar un bloque existente, o incluso todos los bloques de nuestra aplicación? Ese es el trabajo de un plugin de bloques, y Eso a continuación.

Chapter 27: Bloquear Plugins

¡Míranos! Hemos llegado al último tema del tutorial. Ya hemos transformado nuestro sitio estático en uno en el que podemos reordenar el diseño de cada página, mezclarlo con código personalizado de plantillas Twig y añadir contenido dinámico. Eso es... algo impresionante. Por supuesto, no hemos cubierto todo lo que puedes hacer con los Diseños, pero ahora eres realmente peligroso.

¿Crear un bloque personalizado?

Un tema que no hemos tratado es cómo crear un bloque totalmente nuevo, pero está documentado y, a estas alturas, creo que no sería demasiado difícil. ¿Por qué deberías crear un bloque personalizado? Supongamos que tienes algo superpersonalizado como nuestra área "Héroe" o esta área "suscribirse al boletín", que en realidad está potenciada por el paquete UX Live Component de Symfony, que le da el elegante comportamiento Ajax.

En cualquier caso, si quieres algo así en tu página, la forma más sencilla de añadirlo es... como hice yo en este proyecto: poner la lógica en Symfony, renderizar dentro de un bloque Twig, y luego incluir ese bloque Twig dentro de Layouts.

¿Pero qué pasa si queremos que el usuario administrador pueda añadir esto a varias páginas cuando quiera? Entonces sería útil crear un bloque personalizado. Los bloques personalizados también pueden tener opciones, así que incluso podrías permitirles personalizarlo de alguna manera.

Plugins del Bloque Hola

De todos modos, hagamos un último reto relacionado con los bloques: crear un plugin de bloques. Ve a una página de demostración de habilidades. Hmm, probablemente nos vendría bien un poco más de margen entre estos bloques. Y esa es una necesidad bastante común. Podríamos manejar esto añadiendo una clase CSS que establezca el margen. Pero quiero hacerlo aún más fácil.

Ve al administrador de Diseños y edita el Diseño de habilidad individual. Bien, supongamos que queremos añadir algo de margen aquí. Para ello, quiero que el usuario administrador pueda hacer clic en cualquier bloque del sistema -por ejemplo, este bloque de columnas- y, en la pestaña de diseño, seleccionar el margen superior o inferior que necesite de un nuevo campo de formulario.

Es un objetivo bastante descabellado... ¡porque, para conseguirlo, necesitamos poder modificar todos los bloques del sistema! Afortunadamente, ése es exactamente el objetivo de un plugin de bloque: ampliar uno -o todos- los bloques.

Crear el complemento de bloque

Manos a la obra. En el directorio `src/Layouts/`, crea una nueva clase PHP llamada, qué tal, `VerticalWhitespacePlugin`. Tiene que implementar `PluginInterface`. Pero en la práctica, extendemos una clase `Plugin` que implementa esa interfaz por nosotros. Ve a "Código"->"Generar", o `Command`+`N` en un Mac, e implementa el único método que necesitamos: `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.
    }
}
```

Vale, cada bloque del sistema -es decir, cada elemento de aquí del menú de la izquierda- tiene una clase detrás llamada manejador de bloques. Nuestro trabajo en `getExtendedHandlers()` es devolver un iterable de todos los "manejadores" que queramos extender. Por ejemplo, si sólo quisieras extender el bloque del título, podrías `yield TitleHandler::class`. ¿Cómo sabía que debía utilizar esa clase? Bueno, la mayoría de las veces puedes adivinarlo: el bloque de título tiene un `TitleHandler`. Pero si quieres mirar más a fondo, puedes ver todos los manejadores del sistema ejecutando:

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

De todos modos, en nuestro caso, queremos anular cada bloque. Así que podemos `yield BlockHandlerDefinitionInterface::class`, porque cada manejador de bloque debe implementar esa interfaz:

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;
    }
}
```

Y sí, acabo de olvidar por completo la palabra `Definition`. ¡Uy! Arreglaré esta mala interfaz en un minuto.

Añadir un parámetro/campo de bloque personalizado

Para saber qué hacer a continuación, vuelve al menú "Código"->"Generar", selecciona "Anular métodos" y elige `buildParameters()`. No necesitamos llamar al método padre porque está vacío:

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
    }
}
```

Parámetro es la palabra que utiliza Layouts para las opciones del formulario que puedes personalizar en la parte derecha de la pantalla para cada bloque. Gracias a nuestro método `getExtendedHandlers()`, cuando Layouts construya esas opciones para cualquier bloque, ahora llamará a este método y podremos añadir nuevos parámetros.

También necesitamos una declaración `use` para este espacio de nombres `ParameterType`:

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],
            ],
        );
    }
}
```

¡Genial! Como puedes ver, Layouts viene con un montón de "tipos de campo" incorporados, como `BooleanField`, que se mostrará como una casilla de verificación. Su valor predeterminado es falso y tiene una etiqueta. Ah, ¿y este grupo? ¿Recuerdas que hay dos pestañas: "Diseño" y "Contenido"? Aquí es donde determinas dentro de cuál debe vivir tu parámetro.

Y la primera clave - `vertical_whitespace:enabled` es el nombre interno de este campo. Verás cómo lo utilizamos en un minuto.

Antes de que lo intentemos, el futuro Ryan acaba de informarme de que... ¡he metido la pata! Típico, desplázate hacia arriba. ¡Estoy cediendo la clase equivocada! Rendimiento `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
}
```

Así está mejor.

Ahora vamos a probar. Actualiza... haz clic en cualquier bloque... déjame encontrar mi bloque Título... y... ¡ahí está! ¡En cualquier bloque vemos el nuevo campo!

Añadir parámetros/campos "hijos

Pero, la idea real es que, si el usuario lo activa, le mostramos dos campos más donde puede seleccionar el margen superior o inferior.

Para ello, después del primer campo, pegaré dos parámetros más:

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],
            ],
        );
    }
}
```

Estos son básicamente como el primero. La gran diferencia es que, aquí arriba, dijimos `$builder->add()`. Pero ahora tenemos `$builder->get('vertical_whitespace:enabled')` y luego `->add()`. Esto hace que estos campos sean hijos del primero.

Esto está muy bien. Actualiza y... vamos a buscar el bloque Columna. Haz clic en "Activar espacio en blanco vertical". ¡Guau! ¡Aparecen los otros dos campos! Hagamos un espaciado superior "Medio" y un espaciado inferior "No". Publícalo.

Utilizar los parámetros de la plantilla de bloque

Sin embargo, no debería sorprenderte demasiado que cuando actualicemos la página... ¡no ocurra absolutamente nada! Hemos añadido esas opciones... pero aún no las estamos utilizando en ningún sitio. Para ello, necesitamos anular una plantilla.

Pensemos: queremos que este margen superior e inferior se aplique a todos los bloques del sistema. Y, afortunadamente, todos los bloques del sistema acaban extendiéndose a `block.html.twig`: éste de aquí, en el directorio `nglayouts/themes/`.

Cópialo. Luego, anúlalo a través del sistema temático. Si seguimos la ruta... `standard/block`... `standard/block`... el nuevo archivo debería vivir aquí: `block.html.twig`. Pega el contenido dentro.

Para asegurarte de que funciona, pon un poco de `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! Actualiza el frontend. ¡Yupi! Sí, definitivamente funciona. Vamos... quita eso.

En la parte superior de la plantilla, tenemos una variable llamada `css_class`, que está establecida en algunas clases principales. Y ¡eh! ¡Llama a `block.parameter('css_class')`! Sí, ¡eso es lo que lee el campo "Clase CSS" de las opciones del bloque!

Luego, utiliza `|join(' ')` para combinar todo esto en una cadena.

Voy a eliminar ese `join()`... y luego cambiaré el nombre de esta variable a `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
```

Estamos configurando las cosas para que podamos modificar fácilmente esa variable. Aquí abajo, justo antes de `block_content`, vuelve a crear esa variable `css_class` configurada como `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
```

Esta variable se utiliza en un montón de sitios diferentes y en plantillas hijo, así que tenemos que asegurarnos de que sigue establecida.

De todas formas, aquí arriba, ahora tenemos una matriz `css_classes`. ¡Vamos a utilizarla! Voy a pegar tres variables, cada una ajustada al valor de nuestros tres parámetros:

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
```

Aquí es donde resulta útil el nombre del parámetro que utilizamos en la clase.

Ahora, muy sencillo, si `use_whitespace`, entonces añade algunas clases de margen. También pegaré ese código:

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
```

Así, para el margen superior, añadiremos un nuevo `whitespace-top-` seguido de `none`, `small`, `medium` o `large`. Y lo mismo para el inferior.

Estas nuevas clases son totalmente inventadas: no forman parte del CSS de Bootstrap ni de nada, pero podrías hacer esto más inteligente para reutilizarlas. Pero para nosotros, si abres `assets/styles/app.css`... cerca de la parte superior, ¡allá vamos!

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
```

Antes del tutorial, ya he preparado esas clases.

Así que... ¡debería funcionar! Muévete y actualiza. ¡Ya está! Nuestro bloque tiene un pequeño espacio en blanco superior extra... que proviene de nuestra nueva clase.

Y... ¡listo!, ¡Woo! ¡Gran trabajo, equipo! ¡Ahora eres un campeón de Layouts! Hacednos saber qué cosas chulas estáis construyendo con él. Y si tienes alguna pregunta, como siempre, estamos a tu disposición en la sección de comentarios.

Muy bien, gracias y hasta la próxima.

With <3 from SymfonyCasts