Formularios

En este tema se presupone que ya sabemos:

- Construir clases de formulario para formularios estáticos.
- Renderizar formularios
- Configurar validación de datos

En los vídeos veremos formularios embebidos y otros tipos de formularios dinámicos.

Nuevo en Symfony 4.1 y 4.2

Conviene echar un vistazo a un par de novedades sobre renderizdo de formularios y sobre validación de email introducidos en las nuevas versiones:

- Form Field Help: https://symfony.com/blog/new-in-symfony-4-1-form-field-help

- HTML5 Email Validation: https://symfony.com/blog/new-in-symfony-4-1-html5-email-validation

Formularios anidados

Pongamos un ejemplo en el que estamos editando o dando de alta un grado y queremos en esa misma página, editar las asignaturas de dicho grado.

La primera tarea es crear un Formulario para Grado que incluya las asignaturas. Esto se hace añadiendo un campo de tipo **CollectionType**.

```
<?php

namespace App\Form;

use App\Entity\Grado;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\CollectionT
ype;

class GradoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builde
r, array $options)
    {
        $builder
            ->add('nombre')
```

```
        ;

        $builder->add('asignaturas', CollectionType::class
, array(
            'entry_type' => AsignaturaType::class,
            'entry_options' => array('label' => false),
            'allow_add' => true,
        ));
    }

    public function configureOptions(OptionsResolver $reso
lver)
    {
        $resolver->setDefaults([
            'data_class' => Grado::class,
        ]);
    }
}
```

Para definir *CollectionType* hemos de indicar la clase de formulario embebida. También vamos a permitir en este ejemplo añadir nuevos elementos al array CollectionType.

El resto de posibilidades de configuración de CollectionType las podéis ver en este enlace:

https://symfony.com/doc/current/reference/forms/types/collection.html

En nuestro controlador, conseguimos el grado y lo pasamos al formulario, nada nuevo.

```
    /**
     * @Route("/grado/edit", name="grado_edit")
     */
    public function edit(Request $request, GradoRepository
 $gradoRepository)
    {

        $grado = $gradoRepository->find(1);
        $form = $this->createForm(GradoType::class, $grado
);

        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            //
        }

        return $this->render('grado/edit.html.twig', [
            'form' => $form->createView(),
        ]);
    }
```

En twig, renderizamos el formulario.

```
{{ form(form) }}
```

O bien

```
{{ form_start(form) }}

{{ form_row(form.nombre) }}

<h3>Asignaturas</h3>
<ul class="asignaturas">
    {% for asignatura in form.asignaturas %}
        <li style="margin-bottom: 30px;">
          {{ form_row(asignatura.codigo) }}
          {{ form_row(asignatura.nombre) }}
          {{ form_row(asignatura.credects) }}
        </li>
    {% endfor %}
</ul>

{{ form_end(form) }}
```

Podemos observar ya como aparece el formulario del grado y un formulario de asignatura por cada una de las asignaturas que tiene el grado.

Añadir nuevos elementos

Si queremos añadir nuevos elementos, necesitaremos algo de javascript.

Symfony nos ayuda un poco, proporcionándonos el html del formulario de una asignatura nueva, pero el trabajo de javascript lo tenemos que realizar nosotros.

Al poner

```
'allow_add' => true,
```

Symfony crea una variable **prototype** con el html de un formulario de una asignatura nueva.

Podemos guardarnos ese código html en un atributo de algún elemento correctamente escapado para ser utilizado por javascript, por ejemplo:

```
<ul class="asignaturas"
data-prototype="{{ form_widget(form.asignaturas.vars.proto
type)|e('html_attr') }}">
```

Luego nos creamos nuestro javascript a nuestro gusto. Por ejemplo, utilizando JQuery podría ser algo así:

```
<script>
```

```
var $collectionHolder;

var $addAsignaturaLink = $('<a href="#" class="add_asignat
ura_link">Añadir asignatura</a>');
var $newLinkLi = $('<li></li>').append($addAsignaturaLink)
;

jQuery(document).ready(function() {

    $collectionHolder = $('ul.asignaturas');
    $collectionHolder.append($newLinkLi);

    $collectionHolder.data('index', $collectionHolder.find
('li').length);

    $addAsignaturaLink.on('click', function(e) {

        // prevent the link from creating a "#" on the URL
        e.preventDefault();

        // llamar a la función que añade el formulario de
asignatura
        addAsignaturaForm($collectionHolder, $newLinkLi);
    });
});


function addAsignaturaForm($collectionHolder, $newLinkLi)
```

```
{
    // Get the data-prototype explained earlier
    var prototype = $collectionHolder.data('prototype');

    // get the new index
    var index = $collectionHolder.data('index');

    var newForm = prototype;
    // You need this only if you didn't set 'label' => fal
se in your tags field in TaskType
    // Replace '__name__label__' in the prototype's HTML t
o
    // instead be a number based on how many items we have
    // newForm = newForm.replace(/__name__label__/g, index
);

    // Replace '__name__' in the prototype's HTML to
    // instead be a number based on how many items we have
    newForm = newForm.replace(/__name__/g, index);

    // increase the index with one for the next item
    $collectionHolder.data('index', index + 1);

    // Display the form in the page in an li, before the "
Add a tag" link li
    var $newFormLi = $('<li></li>').append(newForm);
    $newLinkLi.before($newFormLi);
}
```

```
</script>
```

Y listo.

Eventos de Formulario

El componente de formularios provee de un proceso estructurado que permite personalizar los formularios a través del EventDispatcher, es decir, a través de eventos.

La siguiente tabla muestra la lista de eventos generados por los formularios en el orden en que son generados:

Name	FormEvents Constant	Event's Data
form.pre_set_data	FormEvents::PRE_SET_DATA	Model data
form.post_set_data	FormEvents::POST_SET_DATA	Model data
form.pre_bind	FormEvents::PRE_SUBMIT	Request data
form.bind	FormEvents::SUBMIT	Normalized data
form.post_bind	FormEvents::POST_SUBMIT	View data

En los siguientes vídeos, vamos a utilizar estos eventos para programar formularios dinámicos.

Formulario dinámico basado en los propios datos del formulario

Para este ejemplo vamos a partir de un formulario de producto con 3 campos: código, nombre y precio.

```
<?php

namespace App\Form;

use App\Entity\Producto;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ProductoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builde
r, array $options)
    {
        $builder
            ->add('codigo')
            ->add('nombre')
            ->add('precio')
        ;
```

```
    }

    public function configureOptions(OptionsResolver $reso
lver)
    {
        $resolver->setDefaults([
            'data_class' => Producto::class,
        ]);
    }
}
```

Y lo que queremos conseguir es que una vez dado de alta un producto, no se pueda cambiar el código.

Lo vamos a hacer con eventos.

Quitamos el campo *codigo* del formulario y lo añadiremos mediante un evento solamente si el objeto producto es nuevo.

Los formularios permite registrar listeners de forma muy sencilla:

```
// src/Form/Type/ProductType.php
namespace App\Form\Type;

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

```
class ProductoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builde
r, array $options)
    {
        $builder
        ->add('nombre')
        ->add('precio')

        $builder->addEventListener(FormEvents::PRE_SET_DAT
A, function (FormEvent $event) {
            // ... código del listener
        });
    }

    // ...
}
```

El código del listener sería algo así:

```
// ...
public function buildForm(FormBuilderInterface $builder, a
rray $options)
{
    // ...
    $builder->addEventListener(FormEvents::PRE_SET_DATA, f
```

```
unction (FormEvent $event) {
        $producto = $event->getData();
        $form = $event->getForm();

        if (!$producto || null === $producto->getId()) {
            $form->add('codigo', TextType::class);
        }
    });
}
```

Con un suscriber también hubiera sido igual de sencillo:

```
// src/Form/Type/ProductType.php
namespace App\Form\Type;

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

class ProductoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builde
r, array $options)
    {
        ->add('nombre')
        ->add('precio')
```

```
        $builder->addEventSubscriber(new MiSuscriber());
    }

    // ...
}
```

Y en el subscriber sería como sigue:

```
namespace App\Form\EventListener;

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\EventDispatcher\EventSubscriberInter
face;
use Symfony\Component\Form\Extension\Core\Type\TextType;

class MiSuscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(FormEvents::PRE_SET_DATA => 'preSetDa
ta');
    }

    public function preSetData(FormEvent $event)
    {
        $producto = $event->getData();
```

```
        $form = $event->getForm();

        if (!$producto || null === $producto->getId()) {
            $form->add('nombre', TextType::class);
        }
    }
}
```

Formulario dinámico basado en información del usuario

En este ejemplo vamos a considerar que nuestra aplicación es algún tipo de red social y que tenemos un formulario para enviar un mensaje a algún amigo, con un desplegable de los amigos del usuario.

La mayor complejidad es acceder al usuario actual. Por suerte en symfony contamos con inyección de dependencias. Podemos inyectar el servicio Security en el Formulario.

Inyectamos el servicio.

```
// src/Form/Type/FriendMessageFormType.php
namespace App\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\TextareaTyp
e;
use Symfony\Component\Security\Core\Security;

class FriendMessageFormType extends AbstractType
```

```
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function buildForm(FormBuilderInterface $builde
r, array $options)
    {
        $builder
            ->add('subject', TextType::class)
            ->add('body', TextareaType::class)
        ;
        $builder->addEventListener(FormEvents::PRE_SET_DAT
A, function (FormEvent $event) {
            // ... add a choice list of friends of the cur
rent application user
        });
    }
}
```

Accedemos al objeto *user* y utilizamos el listener para crear un desplegable con los amigos del usuario.

```
// src/Form/Type/FriendMessageFormType.php
```

```
use App\Entity\User;
use Doctrine\ORM\EntityRepository;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\TextareaTyp
e;
use Symfony\Component\Security\Core\Security;
// ...

class FriendMessageFormType extends AbstractType
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function buildForm(FormBuilderInterface $builde
r, array $options)
    {
        $builder
            ->add('subject', TextType::class)
            ->add('body', TextareaType::class)
        ;

        // grab the user, do a quick sanity check that one
```

```
 exists
        $user = $this->security->getUser();
        if (!$user) {
            throw new \LogicException(
                'The FriendMessageFormType cannot be used
without an authenticated user!'
            );
        }

        $builder->addEventListener(
            FormEvents::PRE_SET_DATA,
            function (FormEvent $event) use ($user) {
                $form = $event->getForm();

                $formOptions = array(
                    'class'         => User::class,
                    'choice_label'  => 'fullName',
                    'query_builder' => function (EntityRep
ository $userRepository) use ($user) {
                        // build a custom query
                        // return $userRepository->createQ
ueryBuilder('u')->addOrderBy('fullName', 'DESC');

                        // or call a method on your reposi
tory that returns the query builder
                        // return $userRepository->createO
rderByFullNameQueryBuilder();
                    },
```

```
                );

                // create the field, this is similar the $
builder->add()
                // field name, field type, data, options
                $form->add('friend', EntityType::class, $f
ormOptions);
            }
        );
    }

    // ...
}
```

Creación de un tema personalizado para formularios

Built-in themes

Symfony 4 viene preparado con varios *built-in themes*. Estos temas son:

- form_div_layout.html.twig
- form_table_layout.html.twig
- bootstrap_3_layout.html.twig
- bootstrap_3_horizontal_layout.html.twig
- bootstrap_4_layout.html.twig
- bootstrap_4_horizontal_layout.html.twig
- foundation_5_layout.html.twig

Por ejemplo, si quisiéramos utilizar Bootstrap 4 en nuestros formularios, el primer paso sería incluir las librerías:

```
{# templates/base.html.twig #}
```

```
{# beware that the blocks in your template may be named di
fferent #}
{% block head_css %}
    <!-- Copy CSS from https://getbootstrap.com/docs/4.0/g
etting-started/introduction/#css -->
{% endblock %}
{% block head_js %}
    <!-- Copy JavaScript from https://getbootstrap.com/doc
s/4.0/getting-started/introduction/#js -->
{% endblock %}
```

Nota: En otro tema veremos cómo utilizar *Webpack Encore* para gestionar nuestros assets.

Y el siguiente paso es configurar twig para que utilice bootstrap en los formularios:

```
# config/packages/twig.yaml
twig:
    form_themes: ['bootstrap_4_layout.html.twig']
```

En vez de configurar un tema de forma global, se puede configurar un tema en un único formulario mediante el tag **form_theme** de Tiwg.

```
{# Esta template solamente aplicaría en este twig #}
{% form_theme form 'bootstrap_4_horizontal_layout.html.twi
g' %}
```

```
{% block body %}
    <h1>User Sign Up:</h1>
    {{ form(form) }}
{% endblock %}
```

Cómo crear un tema personalizado

Para crear un tema personalizado, basta con redefinir uno o más de los mini-bloques (fragments) en los que está divida la plantilla total de un formulario.

Podemos encontrar todos los bloques en form_div_layout.html.twig o en el siguiente enlace:

https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig

Por ejemplo, podemos crear un fichero de twig ({templates/form/fields.html.twig) y definir en dicho fichero el bloque integer_widget que originalmente es así:

```
{% block integer_widget %}
    {% set type = type|default('number') %}
    {{ block('form_widget_simple') }}
{% endblock integer_widget %}
```

Y añadirle una capa *div*

```
{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('form_widget_simple') }}
    </div>
{% endblock %}
```

Ahora le podemos decir a symfony que utilice los bloques que haya definidos en templates/form/fields.html.twig en una plantilla individual

```
{% form_theme form 'form/fields.html.twig' %}
```

o en toda la aplicación

```
# config/packages/twig.yaml
twig:
    form_themes:
        - 'form/fields.html.twig'
    # ...
```

Podemos redefinir tantos bloques como queramos y organizarlos en tantos archivos como queramos. Y añadirlos al form_themes de la configuración o añadirlos con una variante del form_theme en un fichero Twig:

```
{% form_theme form with ['common.html.twig', 'form/fields.
html.twig'] %}
```

Validaciones personalizadas

Como ya sabemos, es muy fácil aplicar reglas de validación (*constraints*) a campos de formularios:

```
public function buildForm(FormBuilderInterface $builder, a
rray $options)
{
    $builder
        ->add('myField', TextType::class, array(
            'required' => true,
            'constraints' => array(new Length(array('min'
=> 3)))
        ))
    ;
}
```

Conviene revisar todos los validadores incluídos en Symfony, es muy posible que la validación que queramos implementar ya lo esté:

https://symfony.com/doc/current/validation.html#constraints

En este vídeo, no obstante, se aborda la creación de reglas de validación personalizadas.

Cómo crear una validación

personalizada

A) Crear la clase Constraint

La creación de una constraint personalizada empieza **extendiendo la clase base Constraint**.

En el siguiente ejemplo, vamos a crear un validador que comprueba que un determinado string contiene únicamente caracteres alfanuméricos:

```
// src/Validator/Constraints/ContainsAlphanumeric.php
namespace App\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class ContainsAlphanumeric extends Constraint
{
    public $message = 'The string "{{ string }}" contains an illegal character: it can only contain letters or numbers.';
}
```

La anotación **@Annotation** es necesaria para poder utilizar esta constraint mediante anotaciones en las clases.

B) Crear el validador

El validador debe de ser otra clase con el mismo nombre que la constraint y con sufijo *Validator*. Debe extender de la clase **ConstraintValidator** y debe implementar un método **validate()**

```
// src/Validator/Constraints/ContainsAlphanumericValidator
.php
namespace App\Validator\Constraints;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class ContainsAlphanumericValidator extends ConstraintVali
dator
{
    public function validate($value, Constraint $constrain
t)
    {
        if (!preg_match('/^[a-zA-Z0-9]+$/', $value, $match
es)) {
            $this->context->buildViolation($constraint->me
ssage)
                ->setParameter('{{ string }}', $value)
                ->addViolation();
        }
    }
```

```
}
```

El método *validate()* no debe de devolver ningún valor. Este método se encargará de añadir **violations**. Si el método termina sin provocar ninguna violación, entonces el valor se considerará válido.

$this->context->buildViolation() construye una violación con un mensaje de error y devuelve una instancia de **ConstraintViolationBuilderInterface** que a su vez tiene un método **addViolation()** que añade la *violation* al *context*.

C) Utilizar el validador

```
// src/Entity/AcmeEntity.php
use Symfony\Component\Validator\Constraints as Assert;
use App\Validator\Constraints as AcmeAssert;

class AcmeEntity
{
    // ...

    /**
     * @Assert\NotBlank
     * @AcmeAssert\ContainsAlphanumeric
     */
    protected $name;
```

```
    // ...
}
```

Inyección de dependencias

Nuestro validador es un servicio, con lo que podemos inyectar a su constructor cualquier otro servicio.

Validaciones de clases

Si queremos validar una clase en su conjunto en vez de un valor, hay que crear un método **getTargets()** en la clase Constraint como sigue:

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

Y entonces, nuestro método *validate()* de la clase validadora recibirá una instancia de la clase a validar en vez de recibir el valor a validar.