

0

The Cookbook

Table of Contents

Allowing "new" Tags with the "Formtype"

Attaching Tags to be Removed

Template Modification

Home ()

Documentation (/doc/current/index.html)

The Cookbook (/index.html)

Form (index.html)

How to Embed a Collection of Forms

How to Embed a Collection of Forms

2.7 version

English

visit this page!

https://github.com/symfony/symfony/blob/master/2.7/cookbook/form/form_collections.en.rst

In this entry, you'll learn how to create a form that embeds a collection of many other forms. This could be useful, for example, if you had a Task class and you wanted to add/create/remove many tag objects related to that Task, right inside the same form.

In this entry, it's loosely assumed that you're using Doctrine as your database store. But if you're not using Doctrine (e.g. Propel or just a database connection), it's all very similar. There are only a few parts of this tutorial that really care about "persistences".

If you are using Doctrine, you'll need to add the Doctrine metadata, including the Symfony/Doctrine association mapping definition on the Task's tags property.

First, suppose that each Task belongs to multiple Tag objects. Start by creating a simple Task class:

```
1 // src/Some/TaskBundle/Form/Type/TagType.php
2 namespace Some\TaskBundle\Form\Type;
3
4 use Doctrine\Common\Collections\ArrayCollection;
5
6 class Task
7 {
8     protected $description;
9
10    protected $tags;
11
12    public function __construct()
13    {
14        $this->tags = new ArrayCollection();
15    }
16
17    public function getDescription()
18    {
19        return $this->description;
20    }
21
22    public function setDescription($description)
23    {
24        $this->description = $description;
25    }
26
27    public function getTags()
28    {
29        return $this->tags;
30    }
31 }
```

The ArrayCollection is specific to Doctrine and is basically the same as using an array (but it must be an ArrayCollection if you're using Doctrine).

Now, create a Tag class. As you saw above, a Task can have many Tag objects:

```
1 // src/Some/TaskBundle/Form/Type/TagType.php
2 namespace Some\TaskBundle\Form\Type;
3
4 class Tag
5 {
6     public $name;
7 }
```

The name property is public here, but it can just as easily be protected or private (but then it would need getters and setters methods).

Then, create a form class so that a Tag object can be modified by the user:

```
1 // src/Some/TaskBundle/Form/Type/TagType.php
2 namespace Some\TaskBundle\Form\Type;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\OptionsResolver\OptionsResolver;
7
8 class TagType extends AbstractType
9 {
10    public function buildForm(FormBuilderInterface $builder, array $options)
11    {
12        $builder->add('name');
13    }
14
15    public function configureOptions(OptionsResolver $resolver)
16    {
17        $resolver->setDefaults(array(
18            'data_class' => 'Some\TaskBundle\Entity\Tag',
19        ));
20    }
21
22    public function getForm()
23    {
24        return 'tag';
25    }
26 }
```

With this, you have enough to render a tag form by itself. But since the end goal is to allow the tags of a task to be modified right inside the task form itself, create a form for the Task class.

Notice that you embed a collection of TagType forms using the collection (...) reference forms/types/collection.html field type:

```
1 // src/Some/TaskBundle/Form/Type/TaskType.php
2 namespace Some\TaskBundle\Form\Type;
3
4 use Symfony\Component\Form\AbstractType;
5 use Symfony\Component\Form\FormBuilderInterface;
6 use Symfony\Component\OptionsResolver\OptionsResolver;
7
8 class TaskType extends AbstractType
9 {
10    public function buildForm(FormBuilderInterface $builder, array $options)
11    {
12        $builder->add('description');
13
14        $builder->add('tags', 'collection', array('type' => new TagType()));
15    }
16
17    public function configureOptions(OptionsResolver $resolver)
18    {
19        $resolver->setDefaults(array(
20            'data_class' => 'Some\TaskBundle\Entity\Task',
21        ));
22    }
23
24    public function getForm()
25    {
26        return 'task';
27    }
28 }
```

In your controller, you'll now initialize a new instance of TaskType:

```
1 // src/Some/TaskBundle/Controller/TaskController.php
2 namespace Some\TaskBundle\Controller;
3
4 use Some\TaskBundle\Entity\Task;
5 use Some\TaskBundle\Entity\Tag;
6 use Some\TaskBundle\Form\Type\TaskType;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
9
10 class TaskController extends Controller
11 {
12    public function newTask(Request $request)
13    {
14        $task = new Task();
15
16        // dummy code - this is here just so that the task has some tags
17        // otherwise, this isn't an interesting example
18        $tags = new Tag();
19        $tags->name = 'tag1';
20        $task->getTags()->add($tags);
21        $tags = new Tag();
22        $tags->name = 'tag2';
23        $task->getTags()->add($tags);
24        // and dummy code
25
26        $form = $this->createForm(new TaskType(), $task);
27        $form->handleRequest($request);
28
29        if ($form->isSubmitted()) {
30            // ... maybe do some form processing, like saving the task and tag objects
31        }
32
33        return $this->render('Some\TaskBundle:Task:new.html.twig', array(
34            'form' => $form->createView(),
35        ));
36 }
```

The corresponding template is now able to render both the description field for the task form as well as all the TagType forms for any tags that are already related to this Task. In the above controller, I added some dummy code so that you can see this in action (since a Task has zero tags when first created).

Twig PHP

```
1 {% from Some\TaskBundle/Resources/views/Task/new.html.twig %}
2
3 {% ... %}
4
5 {{ form_start(form) }}
6 {# render the task's only field: description #}
7 {{ form_row(form.description) }}
8
9 <div>tags</div>
10 <div class="tags">
11     {# Iterate over each existing tag and render its only field: name #}
12     {% for tag in form.tags %}
13         {{ form_row(tag.name) }}
14     {% endfor %}
15 </div>
16 {{ form_end(form) }}
17
18 {% ... %}
```

When the user submits the form, the submitted data for the tags field are used to construct an ArrayCollection of Tag objects, which is then set on the tag field of the Task instance.

The tag collection is accessible naturally via `$task->getTags()` and can be persisted to the database or used however you need.

So far, this works great, but this doesn't allow you to dynamically add new tags or delete existing tags. So, while editing existing tags will work great, your user can't actually add any new tags yet.

In this entry, you embed only one collection, but you are not limited to this. You can also embed nested collection as many levels down as you like. But if you use Xdebug in your development setup, you may receive a Notice function nesting level of '180' reached, aborting! error. This is due to the xdebug_max_nesting_level PHP setting, which defaults to 100.

This directive limits recursion to 100 calls which may not be enough for rendering the form in the template if you render the whole form at once (e.g. `form_render(form)`). To fix this you can set this directive to a higher value (either via a php.ini file or via `ini_set('http://php.net/manual/en/function.ini-set.php)`, for example in `app/autoload.php`) or render each form field by hand using `form_row`.

http://symfony.com/doc/current/cookbook/form/form_collections.html

1/4

Allowing "new" Tags with the "Prototype" ¶

Allowing the user to dynamically add new tags means that you'll need to use some JavaScript. Previously you added two tags to your form in the controller. Now let the user add as many tag forms as they need directly in the browser. This will be done through a bit of JavaScript.

The first thing you need to do is to let the form collection know that it will receive an unknown number of tags. So far you've added two tags and the form type expects to receive exactly two, otherwise an error will be thrown: This form should not contain extra fields. To make this flexible, add the `allow_additional` option to your collection field:

```

1 // src/core/foreshoulder/foreshoulder/foreshoulder.php
2
3 // ...
4 use Symfony\Component\Form\FormBuilderInterface;
5
6 public function buildForm(FormBuilderInterface $builder, array $options)
7 {
8     $builder->add('description');
9
10    $builder->add('tags', 'collection', array(
11        'type'           => new TagType(),
12        'allow_add'      => true,
13    ));
14 }

```

In addition to telling the field to accept any number of submitted objects, the `allow_add` also makes a "prototype" variable available to you. This "prototype" is a little "template" that contains all the HTML to be able to render any new "tag" forms. To render it, make the following change to your template:

```

1 <ul class="tags" data-prototype="{< form_widget(form.tags.vars.prototype)<}">
2   ...
3 </ul>

```

If you render your whole "tags" sub-form at once (e.g. `form_row(form.tags)`), then the prototype is automatically available on the outer div as the data-prototype attribute, similar to what you see above.

The `form.tags.vars.prototype` is a form element that looks and feels just like the individual `form_widget(tag)` elements inside your `for` loop. This means that you can call `form_widget`, `form_row` or `form_label` on it. You could even choose to render only one of its fields (e.g. the `name` field):

```
1  {{ form_widget(form.tags.vars.prototype.name)|e }}
```

On the rendered page, the result will look something like this:

```
1  <div class="tag" data-prototype="<div><div><div class="required" required="" __name__</div></div></div>" id="task_tag">
```

The goal of this section will be to use JavaScript to read this attribute and dynamically add new tag forms when the user clicks a "Add a tag" link. To make things simple, this example uses jQuery and assumes you have it included somewhere on your page.

Add a script tag somewhere on your page so you can start writing some JavaScript.

First, add a link to the bottom of the "tags" list via JavaScript. Second, bind to the "click" event of that link so you can add a new tag form (addTagForm will be show next):

```

1  var ScatterPlotBuilder;
2
3  // setup an "old" x-axis link
4  // "old" = to be replaced by "new"
5  var ScatterPlotBuilder.prototype.addOldTagLink = add a tag/wp/";
6  ScatterPlotBuilder.prototype.addOldTagLink = add a tag/wp/";
7
8  //jQuery(document).ready(function() {
9  // get the data that holds the collection of tags
10  ScatterPlotBuilder.prototype.getTags = function() {
11
12    // add the "old" x-axis anchor and (1 to the tags of
13    ScatterPlotBuilder.prototype.addOldTagLink = add a tag/wp/";
14
15    // count the current Page inputs we have (e.g., 2), use that as the new
16    // x-axis when inserting a new tag (e.g., 2)
17    ScatterPlotBuilder.prototype.addOldTagLink = add a tag/wp/";
18
19    // prepare the link Page creating a "x" on the URL
20    // www.example.com/?x=1, function() {
21
22    // add a new tag Page (see next code block)
23    ScatterPlotBuilder.prototype.addOldTagLink = add a tag/wp/";
24  }
25  });

```

The `addtagform` function's job will be to use the `data-prototype` attribute to dynamically add a new form when this link is clicked. The `data-prototype` HTML contains the tag text input element with a name of `task[tags][__name__][name]` and id of `task_tags__name__name`. The `__name__` is a little "placeholder", which you'll replace with a unique, incrementing number (e.g. `task[tags][3][name]`).

The actual code needed to make this all work can vary quite a bit, but here's one example:

```

1 function findIndex(a, callback) {
2     // get the auto-prototype explained earlier
3     var prototype = $collectionWrapper.data('prototype');
4
5     // get the new index
6     var index = $collectionWrapper.data('index');
7
8     // Replace __name__ in the prototype's HTML to
9     // instead of a number based on how many items we
10    var newname = prototype.replace(/__name__/g, index);
11
12    // Decrease the index with one for the next item
13    $collectionWrapper.data('index', index + 1);
14
15    // Display the form in the page on an i, before
16    var $newForm1 = $('div[id=i]').append(newForm);
17    $newForm1.before($newForm1);
18 }

```

It is better to separate your JavaScript in real JavaScript files than to write it inside the HTML as is done here.

Now, each time a user clicks the `Add a tag` link, a new sub form will appear on the page. When the form is submitted, any new tag forms will be converted into new `Tag` objects and added to the `tags` property of the `Task` object.

You can find a working example in this JSFiddle (<http://jsfiddle.net/847Xf/4/>).

To make handling these new tags easier, add an "adder" and a "remover" method for the tags in the Task class

```

1 // archive/TaskBuilder/GetTryTask.php
2 namespace Acme\TaskBuilder\Utility;
3
4 // ...
5 class Task
6 {
7     // ...
8
9     public function setTag(Tag $tag)
10     {
11         $this->tag->add($tag);
12     }
13
14     public function removeTag(Tag $tag)
15     {
16         // ...
17     }
18 }

```

Next, add a `by_reference` option to the `tags` field and set it to `false`:

```

1 // src/scene/Taskbundle/Form/Type/TaskType.php
2
3 // ...
4 public function buildForm(FormBuilderInterface $builder, array $options)
5 {
6     // ...
7
8     $builder->add('tags', 'collection', array(
9         // ...
10        'by_reference' => false,
11    ));
12 }

```

With these two changes, when the form is submitted, each new `Tag` object is added to the `Task` class by calling the `addTag` method. Before this change, they were added internally by the form by calling `$task->getTags()->add($tag)`. That was just fine, but forcing the use of the "adder" method makes handling these new `Tag` objects easier (especially if you're using Doctrine, which you will learn about next!).

You have to create both `addTag` and `removeTag` methods, otherwise the form will still use `setTag` even if `by_reference` is false. You'll learn more about the `removeTag` method later in this article.

Doctrine: Cascading Relations and saving the "Inverse" side

To save the new tags with Doctrine, you need to consider a couple more things. First, unless you iterate over all of the new Tag objects and call `saveOrCreate($tag)` on each, you'll receive an error from Doctrine:

A new entity was found through the relationship Acme\FastBundle\Entity\FastTags that was not configured to cascade persist operations for entity...

To fix this, you may choose to "cascade" the persist operation automatically from the Task object to any related tags. To do this, add the cascade option to your ManyToMany metadata:

Annotations YAML XML

```

1 // ...
2
3 // ...
4
5 /**
6  * @param ManyToMany(targetEntity="Tag", cascade={"persist"})
7  */
8 protected $tags;

```

A second potential issue deals with the Owning Side and Inverse Side (<http://docs.doctrine-project.org/en/latest/reference/unitofwork-associations.html>) of Doctrine relationships. In this example, if the "owning" side of the relationship is "Task", then persistence will work fine as the tags are properly added to the Task. However, if the owning side is on "User" then we'll need to do a little bit more work to ensure that the correct side of the relationship is modified.

The trick is to make sure that the single "Task" is set on each "Tag". One easy way to do this is to add some extra logic to `addTag()`, which is called by the form type since by reference is set to false:

```
1 // src/scene/TaskBundle/Entity/Task.php
2
3 // ...
4 public function addTag(Tag $tag)
5 {
6     $tag->addTask($this);
7
8     $this->tags->add($tag);
9 }
10
```

```
1 // src/Some/TaskBundle/Entity/Tag.php
2
3 // ...
4 public function addTask(Task $task)
5 {
6     if ($this->tasks->contains($task)) {
7         $this->tasks->add($task);
8     }
9 }
```

If you have a one-to-many relationship, then the workaround is similar, except that you can simply call `setTask` from inside `addTag`.

Allowing Tags to be Removed ¶

The next step is to allow the deletion of a particular item in the collection. The solution is similar to allowing tags to be added.

Start by adding the `allow_remove` option in the form type:

```
1 // src/Some/TaskBundle/Form/Type/TaskType.php
2
3 // ...
4 public function buildForm(FormBuilderInterface $builder, array $options)
5 {
6     // ...
7     $builder->add('tags', 'collection', array(
8         // ...
9         'allow_remove' => true,
10     ));
11 }
```

Now, you need to put some code into the `removeTag` method of `Task`:

```
1 // src/Some/TaskBundle/Entity/Task.php
2
3 // ...
4 class Task
5 {
6     // ...
7
8     public function removeTag(Tag $tag)
9     {
10         $this->tags->removeElement($tag);
11     }
12 }
```

Template Modifications ¶

The `allow_remove` option has one consequence: if an item of a collection isn't sent on submission, the related data is removed from the collection on the server. The solution is thus to remove the form element from the DOM.

First, add a "delete this tag" link to each tag form:

```
1 <script>
2
3 // ...
4
5 // add a delete link to all of the existing tag form elements
6 $(function() {
7     addTagFormDeleteLink($('tag'));
8 });
9
10 // ... the rest of the block from above
11
12
13
14
15 function addTagForm() {
16     // ...
17
18     // add a delete link to the new form
19     addTagFormDeleteLink($newForm);
20 }
```

The `addTagFormDeleteLink` function will look something like this:

```
1 function addTagFormDeleteLink($tagForm) {
2     var $newForm = $('<div>New Form</div>').clone().html('');
3     $tagForm.prepend($newForm);
4
5     $newForm.on('click', function() {
6         // prevent the link from creating a "x" on the click
7         e.preventDefault();
8
9         // remove the li for the tag form
10         $tagForm.remove();
11     });
12 }
```

When a tag form is removed from the DOM and submitted, the removed tag object will not be included in the collection passed to `writeTag`. Depending on your persistence layer, this may or may not be enough to actually remove the relationship between the removed tag and `Task` object.

Doctrine: Ensuring the database persistence

When removing objects in this way, you may need to do a little bit more work to ensure that the relationship between the `Task` and the removed tag is properly removed.

In Doctrine, you have two sides of the relationship: the owning side and the inverse side. Normally in this case you'll have a many-to-many relationship and the deleted tags will disappear and persist correctly (adding new tags also works effortlessly).

But if you have a one-to-many relationship or a many-to-many relationship with a dependency on the `Task` entity (meaning `Task` is the "inverse" side), you'll need to do more work for the removed tags to persist correctly.

In this case, you can modify the controller to remove the relationship on the removed tag. This assumes that you have some extractor which is handling the "update" of your `Task`.

```
1 // src/Some/TaskBundle/Controller/TaskController.php
2
3 use Doctrine\Common\Collections\ArrayCollection;
4
5 // ...
6 public function editAction($id, Request $request)
7 {
8     $task = $this->getEntityManager()->getRepository('Some\TaskBundle:Task')->find($id);
9     $tags = $task->getTags();
10
11     if ($tags) {
12         throw $this->createNotFoundException('No task found for id '.$id);
13     }
14
15     $existingTags = new ArrayCollection();
16
17     // Create an ArrayCollection of the current tag objects in the database
18     foreach ($tags->getValues() as $tag) {
19         $existingTags->add($tag);
20     }
21
22     $editForm = $this->createForm(new TaskType(), $task);
23     $editForm->handleRequest($request);
24
25     if ($editForm->isSuccessful()) {
26         // remove the relationship between the tag and the task
27         foreach ($existingTags as $tag) {
28             if ($task == $task->getTag()->getTask($tag)) {
29                 // remove the task from the tag
30                 $tag->getTask()->removeElement($task);
31
32                 // If it is a many-to-many relationship, remove the relationship like this
33                 // $tag->setTask(null);
34
35                 $task->remove($tag);
36             }
37         }
38
39         // If you wanted to delete the tag entirely, you can also do that
40         // $task->remove($tag);
41     }
42
43     $task->persist($task);
44     $task->flush();
45
46     // redirect back to new edit page
47     return $this->render($task->getForm(), array('id' => $id));
48 }
49
50 // render new form template
51 }
```

As you can see, adding and removing the elements correctly can be tricky. Unless you have a many-to-many relationship where `Task` is the "owning" side, you'll need to do extra work to make sure that the relationship is properly updated whether you're adding new tags or removing existing tags on each `Tag` object itself.

This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Europe License (<http://creativecommons.org/licenses/by-sa/3.0/>)

News from the Symfony blog

- New in Symfony 2.8: Redesigning Web Debug Toolbar
August 01, 2015
([Blog: new-in-symfony-2-8-redesigning-web-debug-toolbar](#))
 - A week of symfony #448 (27 July - 2 August 2015)
August 01, 2015
([Blog: a-week-of-symfony-448-27-july-2-august-2015](#))
 - Symfony 2.7.3 released
July 31, 2015
([Blog: symfony-2-7-3-released](#))
- Visit The Symfony Blog (Blog)

Do the news



<http://sensiolabs.com/en/symfony/certification.html>
Symfony Certification: Now in 4,000 centers around the world!
GET CERTIFIED (<http://sensiolabs.com/en/symfony/certification/order>)

Enjoying training sessions
Web Development with Symfony2
... ..

06/08/2015

How to Embed a Collection of Forms (The Symfony Cookbook)

http://training.semanticslabs.com/en/training/courses?search_training%3Bsearch%3BfromSD=4&search_training%3Bavailable_languages%3BDS805SD=f&search_training%3Bavailable_languages%3BDS81SD=ind&search_training%3Bavailable_languages%3BDS82SD=de&search_training%3Bavailable_languages%3BDS83SD=de&search_training%3Bkeyword%3B=ONC%3Bdevelopment+Web+ave+Symfony&search_training%3BfromSD=0&11/201&search_training%3BfromSD=0&14/2015
 Getting Started with Symfony2
 London - 2015-08-13
http://training.semanticslabs.com/en/training/courses?search_training%3Bsearch%3BfromSD=2&search_training%3Bavailable_languages%3BDS805SD=f&search_training%3Bavailable_languages%3BDS81SD=ind&search_training%3Bavailable_languages%3BDS82SD=de&search_training%3Bavailable_languages%3BDS83SD=de&search_training%3Bkeyword%3B=ONC%3Bdriver+ave+Symfony&search_training%3BfromSD=0&11/201&search_training%3BfromSD=0&12/2015
 Mastering Symfony2
 London - 2015-08-13
http://training.semanticslabs.com/en/training/courses?search_training%3Bsearch%3BfromSD=2&search_training%3Bavailable_languages%3BDS805SD=f&search_training%3Bavailable_languages%3BDS81SD=ind&search_training%3Bavailable_languages%3BDS82SD=de&search_training%3Bavailable_languages%3BDS83SD=de&search_training%3Bkeyword%3B=ONC%3Bdriver+ave+Symfony&search_training%3BfromSD=0&11/201&search_training%3BfromSD=0&14/2015
 View all sessions (<http://training.semanticslabs.com>)

is a trademark of Fabien Potencier. All rights reserved.

The Cookbook