4. Forms

We are going to be working on the repository that you have created up to this point. If you skipped ahead, don't worry. You can clone the repository from its Github repository.

```
$ git clone git@github.com:skafandri/symfony-tutorial.git --branch ch3
Cloning into 'symfony-tutorial'...
remote: Counting objects: 443, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 443 (delta 2), reused 0 (delta 0), pack-reused 422
Receiving objects: 100% (443/443), 226.44 KiB | 343.00 KiB/s, done.
Resolving deltas: 100% (203/203), done.
Checking connectivity... done.
```

Don't forget to `composer install`.

4.1 Order

4.1.1 Status

We will start by defining an initial set of possible status values for an order (new, processing, delivered, cancelled)

- Edit **src/AppBundle/Entity/Order.php** and add the following after the class definition

```
const STATUS_NEW = 1;
const STATUS_PROCESSING = 10;
const STATUS_DELIVERED = 20;
const STATUS_CANCELLED = 30;
```

Since we have numerical values, we will map them strings as translation messages.

- Create **src/AppBundle/Resources/translations/messages.en.yml**

```
order:
    status:
        1: new
        10: processing
        20: delivered
        30: cancelled
```

We will update the order lsiting template to show the status as string values.

- Edit **src/AppBundle/Resources/views/order/index.html.twig**

```
{% extends 'common/app.html.twig' %}

{% block header%}
    <h1>Order list</h1>
{% endblock %}

{% block content%}
    <table class="records_list">
        <thead>
            <tr>
                <th>Id</th><th>Status</th><th>Createdate</th><th>Actions</th>
            </tr>
        </thead>
        <tbody>
            {% for entity in entities %}
              <tr>
                  <td>
                    <a href="{{ path('order_show', { 'id':entity.id }) }}">
                        {{ entity.id }}
                    </a>
                  </td>
                  <td>
                    {{ ('order.status.' ~ entity.status)|trans }}
                  </td>
                  <td>
                    {% if entity.createDate %}
                      {{ entity.createDate|date('Y-m-d H:i:s') }}
                    {% endif %}
                  </td>
                  <td>
                    <div class="actions">
                      <a class="button"
                        href="{{ path('order_show', { 'id': entity.id }) }}">
                          show
                      </a>
                      <a class="button"
                        href="{{ path('order_edit', { 'id': entity.id }) }}">
                          edit
                      </a>
                    </div>
                  </td>
              </tr>
            {% endfor %}
        </tbody>
    </table>
    <a class="button" href="{{ path('order_new') }}">
        New Order
    </a>
{% endblock %}
```

We need to enable the translation in our application configuration

- Edit **app/Resources/config/config.yml** and uncomment `translator: { fallbacks: ["%locale%"] }` under *framework* key (add it if it doesn't exist) and make sure you have `locale: en` in **app/Resources/config/parameters.yml**

A snippet from config.yml:

```
framework:
    translator:      { fallbacks: ["%locale%"] }
    secret:          "%secret%"
```

If you visit http://127.0.0.1:8000/order/ you should see the Status column displaying (new, processing..) if you have correct status values (1,10,20,30)

Next thing we will do, is removing the possibility to enter a status when creating or editing an order.

All we need to do is to edit **src/AppBundle/Form/OrderType.php** and remove the line `->add('status')` from **buildForm** method.

No we want the order status to be automatically set to new when creating a new order. There are many ways to achieve this, we will discuss only 3 of them because of their popularity among programmers implementations.

- Set a default value of *1* to the *order.status* column
  First, we are implementing a business rule in the database server, which is mixing responsibilities and limiting our application's portability for no reason. Second, we are implementing it wrong, because 1 is not a default value (that should be used in case no value is provided) The new status should be set for any any order when created.

- Update `OrderController::createAction` to enforce setting the status to new
  It is very possible that another developer (or the future you) tries to create a similar action in another part of the application and forget to enforce the same logic (how would it be if we add even more logic to the order creation process..).

- Doctrine lifecycle events
  Doctrine triggers several events during it's lifecycle. It enables us to register methods to be called during an entity manager lifecycle.

> To achieve our goal, we will use the **prePersist** event. For a full list of lifecycle events, please visit http://doctrine-orm.readthedocs.org/en/latest/reference/events.html#lifecycle-events

- Edit **src/AppBundle/Entity/Order.php** and add `@ORM\HasLifecycleCallbacks` to the class annotation block

```
/**
 * Order
 * ...
 * ...
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Order
```

At the end of the class, add the following method

```
/**
 * @ORM\PrePersist
 */
public function prePersist()
{
    if (!$this->id) {
        $this->status = self::STATUS_NEW;
    }
}
```

Done, if you create a new order now, it will have the status *new=1*

4.1.2 Add products to order

For the moment there is no way to add products when creating a new order. We will implement this feature.

Doctrine already generated a ManyToOne relation from OrderProductLine entity to Order entity. We need to add a OneToMany relation from Order to OrderProductLine entities.

- Edit **src/AppBundle/Entity/Order.php** and add `$productLines` after `$customer`

```
/**
 * @var \Doctrine\Common\Collections\Collection
 *
 * @ORM\OneToMany(targetEntity="OrderProductLine", mappedBy="order")
 */
private $productLines;
```

Note here that we didn't provided any database details about the relation (table, join column, joined column). Doctrine will get the mapping informations from the target entity's order member `OrderProductLine::order`

Add the following accessors after **getCustomer()**.

```
public function getProductLines()
{
    return $this->productLines;
}

public function setProductLines(array $productLines)
{
    $this->productLines = $productLines;
    return $this;
}

public function addProductLine(OrderProductLine $productLine)
{
    $productLine->setOrder($this);
    $this->productLines[] = $productLine;
    return $this;
}
```

We will need a custom display to add products to the order, we will also add an Ajax autocomplete to lookup products by code.

This will involve a little more effort, basically we will need to

- Create a form type that can be embedded in the OrderType and accept some options.
- Create a new widget to display our form.
- Implement JavaScript logic to interactively fetch products from the server and provide visual feedback to the user.
- Implement server side logic to fetch and return products details by product code
- Implement server side logic to persist a new Order with a collection of SaleProducts

- Create a new FormType **src/AppBundle/Form/OrderProductLineEmbeddableType.php**

```php
<?php

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\Form\FormView;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class OrderProductLineEmbeddableType extends AbstractType
{

    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        $view->vars['title'] = $options['title'];
        $view->vars['route'] = $options['route'];
        $view->vars['columns'] = $options['columns'];
        $view->vars['search'] = $options['search'];
    }

    /**
     * @param OptionsResolverInterface $resolver
     */
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => null,
            'title' => null,
            'route' => null,
            'columns' => array(),
            'search' => null,
            'itemsPerPage' => 10
        ));
    }

    /**
     * @return string
     */
    public function getName()
    {
        return 'appbundle_orderproductline_embeddable';
    }

}
```

- Edit **src/AppBundle/Form/OrderType.php** and update buildForm() method

```
/**
 * @param FormBuilderInterface $builder
 * @param array $options
 */
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
            ->add('customer')
            ->add('productLines', new OrderProductLineEmbeddableType(), array(
                'title' => 'products',
                'route' => 'productsale_search_by_code',
                'columns' => array('code', 'title', 'price', 'quantity'),
                'search' => 'code'
            ))
    ;
}
```

- Create a new widget **src/AppBundle/Resources/views/Form/fields.html.twig**

```
{% block appbundle_orderproductline_embeddable_widget %}
    <div {{ block('widget_container_attributes') }}
        class="ui-widget-content embeddable-form"
        data-source="{{ path(route) }}" data-name="{{ full_name }}">
        <div class="ui-widget-header">
            {{ title }}
        </div>
        search <input type="text" class="search-input" placeholder="{{ search }}"/>
        <table class="data">
            <tr>
                <th></th>
                    {% for column in columns %}
                    <th> {{ column }} </th>
                    {% endfor %}
                <th></th>
            </tr>
        </table>
    </div>
    <script type="text/javascript"
            src="{{asset('bundles/app/js/embeddable-form.js')}}" >
    </script>
{% endblock %}
```

- To register the new widget within form themes, edit **app/config/config.yml**
  add

```
form_themes:
    - 'AppBundle:Form:fields.html.twig'
```

under *twig* key

a snippet from my config.yml

```
# Twig Configuration
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"
    form_themes:
        - 'AppBundle:Form:fields.html.twig'
```

- Edit **src/AppBundle/Controller/ProductSaleController.php**

Add the following uses

```
use AppBundle\Entity\Product;
use Doctrine\ORM\Query\Expr\Join;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
```

Add the following action

```
public function searchByCodeAction(Request $request)
{
    $code = $request->get('term');
    $entityManager = $this->getDoctrine()->getManager();
    $repository = $entityManager->getRepository(ProductSale::REPOSITORY);
    $queryBuilder = $repository->createQueryBuilder('productSale');
    $products = $queryBuilder
            ->select('productSale')
            ->where('product.code like :code')
            ->andWhere('productSale.active=1')
            ->innerJoin(
                    Product::REPOSITORY, 'product', Join::WITH, $queryBuilder
                    ->expr()
                    ->eq('product', 'productSale.product')
            )
            ->setParameter('code', "$code%")
            ->getQuery()
            ->getResult();
    $response = array();
    foreach ($products as $productSale) {
        $response[] = array(
            'id' => $productSale->getId(),
            'code' => $productSale->getProduct()->getCode(),
            'title' => $productSale->getProduct()->getTitle(),
            'price' => $productSale->getPrice(),
        );
    }
    return new JsonResponse($response);
}
```

- Edit **src/AppBundle/Resources/config/routing/productsale.yml** and add a route to the action we just created

```
productsale_search_by_code:
    path:     /search/code
    defaults: { _controller: "AppBundle:ProductSale:searchByCode" }
```

- Edit **src/AppBundle/Resources/public/js/embeddable-form.js**

```
$(function () {
    $('.embeddable-form').each(function () {
        var source = $(this).data('source');
        var name = $(this).data('name');
        var dataTable = $(this).find('.data');
        $(this).find('.search-input').autocomplete({
            source: source,
            minLenth: 2,
            select: function (event, ui) {
                var row = createProductRow(ui.item, name);
                dataTable.append(row);
            }

        }).autocomplete("instance")._renderItem = function (ul, item) {
            return $("<li>").text(item.code).appendTo(ul);
        };
    });
});

function createProductRow(product, inputName) {
    var id = $('<td>').append($('<input type="hidden">')
            .prop('name', inputName + '[' + product.id + ']'));
    var code = $('<td>').html(product.code);
    var title = $('<td>').html(product.title);
    var price = $('<td>').html(product.price);
    var quantity = $('<td>')
      .append('<input type="text" name="quantity[' + product.id + ']" value="1"/>');
    var deleteButton = $('<td>').append($('<div title="delete">').button({
        icons: {
            primary: "ui-icon-trash"
        },
        text: false
    }).on('click', function () {
        $(this).parent().parent().remove();
    }));

    return $('<tr>')
            .append(id)
            .append(code)
            .append(title)
            .append(price)
            .append(quantity)
            .append(deleteButton);
}
```

-Edit **src/AppBundle/Controller/OrderController.php**

Add the follwing uses

```
+use AppBundle\Entity\Customer;
+use AppBundle\Entity\OrderProductLine;
+use AppBundle\Entity\ProductSale;
```

Update createAction() method

```
public function createAction(Request $request)
{
    $entityManager = $this->getDoctrine()->getManager();
    $postOrder = $request->get('appbundle_order');
    $quantities = $request->get('quantity');

    $order = new Order();

    $order->setCustomer(
            $entityManager
                    ->getRepository(Customer::REPOSITORY)
                    ->find($postOrder['customer'])
    );

    foreach ($postOrder['productLines'] as $productSaleId => $value) {
        $productLine = new OrderProductLine();
        $productLine->setProductSale(
                $entityManager
                        ->getRepository(ProductSale::REPOSITORY)
                        ->find($productSaleId)
        );
        $productLine->setQuantity($quantities[$productSaleId]);
        $entityManager->persist($productLine);
        $order->addProductLine($productLine);
    }

    $entityManager->persist($order);
    $entityManager->flush();

    return $this->redirect(
                    $this->generateUrl('order_show', array('id' => $order->getId()))
    );
}
```

- Edit **src/AppBundle/Entity/Customer.php** and add `const REPOSITORY = 'AppBundle:Customer';` after class declaration.
- Edit **src/AppBundle/Entity/Product.php** and add `const REPOSITORY = 'AppBundle:Product';` after class declaration.
- Edit **src/AppBundle/Entity/ProductSale.php** and add `const REPOSITORY = 'AppBundle:ProductSale';` after class declaration.
- Edit **src/AppBundle/Entity/Warehouse.php** and add __toString() method

```
public function __toString()
{
    return $this->getName();
}
```

Visit http://127.0.0.1:8000/order/new and see how it works (make sure you have some test data in your database)

4.1.3 Products in order show view

The order view page doesn't show any products, we will add this feature now.
Before jumping into implementation, let's sit back and think for a while. Listing some sub records from a parent record is a pretty common task, we will eventually need a similar feature somewhere else in our application.
Let's abstract the listing concept with a minimum implementation that we will enhance whenever we need more features.

- Create **src/AppBundle/Resources/views/common/listing-table.html.twig**

```
<table>
    <tr>
        {% for column in model %}
            <th> {{column.title}} </th>
            {% endfor %}
    </tr>
    {% for item in data %}
        <tr>
            {% for column in item %}
                <td>
                    {{ column }}
                </td>
            {% endfor %}
        </tr>
    {% endfor %}
</table>
```

We will include this template from any records listing template, the first one we will create is the product lines listing.

- Create **src/AppBundle/Resources/views/OrderProductLine/listing-table.html.twig**

```
{% set rows = [] %}

{% for productLine in productLines %}
    {% set row = {
        'code': productLine.productSale.product.code,
        'title': productLine.productSale.product.title,
        'price': productLine.productSale.price,
        'quantity': productLine.quantity,
    }%}
    {% set rows = rows|merge({(loop.index0): row})%}
{% endfor %}

{{ include('AppBundle:common:listing-table.html.twig',
    {
        model: [
            {'title': 'code'},
            {'title': 'title'},
            {'title': 'price'},
            {'title': 'quantity'}
        ],
        data: rows
    }

) }}
```

Now we need to update the order view template to include the listing template

- Edit **src/AppBundle/Resources/views/Order/show.html.twig**

```
{% extends 'common/app.html.twig' %}

{% block header %}
    <h1>Order</h1>
{% endblock %}

{% block content %}
    <table class="record_properties">
        <tbody>
            <tr>
                <th>Id</th>
                <td>
                    {{ entity.id }}
                </td>
            </tr>
            <tr>
                <th>Status</th>
                <td>
                    {{ entity.status }}
                </td>
            </tr>
            <tr>
                <th>Createdate</th>
                <td>{{ entity.createDate|date('Y-m-d H:i:s') }}</td>
            </tr>
            <tr>
                <th>Products</th>
                <td>
                    {{ include('AppBundle:OrderProductLine:listing-table.html.twig',
                        {'productLines': entity.productLines}
                    ) }}
                </td>
            </tr>
        </tbody>
    </table>

    <div class="record-actions">
        <a class="button" href="{{ path('order') }}">
            Back to the list
        </a>
        <a class="button" href="{{ path('order_edit',
            { 'id': entity.id }) }}">
            Edit
        </a>
        <div style="float:left">
            {{ form(delete_form) }}
        </div>
        <div style="clear:both"></div>
    </div>
{% endblock %}
```

Done, view an order that has at least one product line and see the result.

4.1.4 Products in order edit view

When trying to edit an order with existing products, we need to see those products.

- Edit **src/AppBundle/Resources/views/Form/fields.html.twig** to show listings if any data is available

```
{% block appbundle_orderproductline_embeddable_widget %}
    {% set rows = [] %}
    {% if (value is not null) %}
        {% for productLine in value.owner.productLines %}
            {% set row = {
                'code': productLine.productSale.product.code,
                'title': productLine.productSale.product.title,
                'price': productLine.productSale.price,
                'quantity': productLine.quantity,
            }%}
            {% set rows = rows|merge({(loop.index0): row})%}
        {% endfor %}
    {% endif %}

    <div {{ block('widget_container_attributes') }}
        class="ui-widget-content embeddable-form"
        data-source="{{ path(route) }}"
        data-name="{{ full_name }}"
        data-rows="{{ rows|json_encode() }}">

        <div class="ui-widget-header">
            {{ title }}
        </div>
        search
        <input type="text" class="search-input" placeholder="{{ search }}"/>
        <table class="data">
            <tr>
                <th></th>
                    {% for column in columns %}
                    <th> {{ column }} </th>
                    {% endfor %}
                <th></th>
            </tr>
        </table>
    </div>
    <script type="text/javascript"
            src="{{asset('bundles/app/js/embeddable-form.js')}}" >
    </script>
{% endblock %}
```

- Edit **src/AppBundle/Resources/public/js/embeddable-form.js**
  Add the following method

```
function appendInitialRows(initialRows, name, dataTable) {
    if (!initialRows) {
        return;
    }
    for (var index in initialRows) {
        var row = createProductRow(initialRows[index], name);
        dataTable.append(row);
    }
}
```

update the onLoad function and add `appendInitialRows($(this).data('rows'), name, dataTable);`

```
$(function () {
    $('.embeddable-form').each(function () {
        var source = $(this).data('source');
        var name = $(this).data('name');
        var dataTable = $(this).find('.data');
        $(this).find('.search-input').autocomplete({
            source: source,
            minLenth: 2,
            select: function (event, ui) {
                var row = createProductRow(ui.item, name);
                dataTable.append(row);
            }

        }).autocomplete("instance")._renderItem = function (ul, item) {
            return $("<li>").text(item.code).appendTo(ul);
        };
        appendInitialRows($(this).data('rows'), name, dataTable);
    });
});
```

Now if you go to and order edit page, you should see it preloaded with the corresponding products.

4.2 Homework

1. Create a dahboard with the path /**dashboard** with links to all the available listings
2. Create a link to the dashboard in all the lsitings footers
3. Update OrderController:updateAction to work as expected
   - Add new products if needed
   - Remove existing products if removed
   - Update the quantity