



The symfony Cookbook

symfony 1.0

This PDF is brought to you by
SENSIOLABS 

License: Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported License
Version: cookbook-1.0-en-2010-05-24

Table of Contents

How to manage a Shopping Cart	3
Overview	3
Installation	3
Constructor	3
Create a User Shopping Cart	4
Add, modify and remove Items	4
The shoppingcart module	5
Add an item	6
Modify an item.....	6
Delete an entire shopping cart.....	7
Display the Shopping Cart in a Template	7
Get the content of the shopping cart.....	7
Pass the values to the template.....	8
With or without Taxes.....	8

Chapter 1

How to manage a Shopping Cart

Overview

Symfony offers a plugin to manage shopping carts in ebusiness websites. Adding an item, changing quantities and displaying the content of a shopping cart is made easy and painless.

Installation

The shopping cart classes are not shipped with the default symfony installation, but packaged into a plugin called `sfShoppingCart`. Symfony plugins are installed via PEAR (find more about plugins in Chapter 17¹).

The installation of the `sfShoppingCart` plugin is very straightforward, as described in the plugin page². You just need to type in the command line:

```
$ symfony plugin-install http://plugins.symfony-project.com/  
sfShoppingCartPlugin
```

*Listing
1-1*

Then clear the cache to enable autoloading on the plugin's classes:

```
$ symfony cc
```

*Listing
1-2*

Constructor

The `sfShoppingCart` class is aimed to manage shopping carts. It can contain any kind of object.

The constructor allows to declare the tax rate to apply to the shopping cart:

```
$my_shopping_cart = new sfShoppingCart(sfConfig::get('app_cart_tax'));
```

*Listing
1-3*

In this example, the shopping cart tax rate is written in the `app.yml` configuration file of the application for easy change:

*Listing
1-4*

1. http://www.symfony-project.org/book/1_0/17-Extending-Symfony#chapter_17_plug_ins
2. `sfShoppingCart`

```
all:
  cart:
    tax: 19.6
```

Create a User Shopping Cart

You can easily create a new `sfShoppingCart` object in an action with the `new` constructor. However, it will not be of any use if it is not linked to a user session. The easiest way to keep a user's choice in a shopping cart is to make a composition³ of an `sfShoppingCart` object into the `sfUser` class. To do that, add a custom method to the `myproject/apps/myapp/lib/myUser.php` class:

```
Listing 1-5 class myUser extends sfUser
{
    public function getShoppingCart()
    {
        if (!$this->hasAttribute('shopping_cart'))
        {
            $this->setAttribute('shopping_cart', new
sfShoppingCart(sfConfig::get('app_cart_tax')));
        }

        return $this->getAttribute('shopping_cart');
    }

    // ...
}
```

The `$user->getShoppingCart()` method will create a new shopping cart if the user doesn't already have one.



if you need more information about the way to override the default `sfUser` class by a custom `myUser` class, you should read the section about **factories** in Chapter 17⁴.

Add, modify and remove Items

The shopping cart can contain any quantity of objects from different classes. Each item stored in the shopping cart is an instance of the `sfShoppingCartItem` class.

The `sfShoppingCart` class has `addItem()` and `deleteItem()` methods. As you can add or delete any type of object, the first argument of these method calls is the class name of the object.

To modify the quantity of one item, first get the `sfShoppingCartItem` object itself (via the `getItems()` method of the `sfShoppingCart` object) and call its `setQuantity()` method.

3. http://en.wikipedia.org/wiki/Object_composition

4. http://www.symfony-project.org/book/1_0/17-Extending-Symfony#chapter_17_factories

The shoppingcart module

Here is a possible module implementation of the shopping cart management where objects of class 'Product' (representing products) can be added, modified or suppressed with the actions 'add', 'update' and 'delete':

```
class shoppingcartActions extends sfActions
{
    // ...

    public function executeIndex()
    {
        $this->shopping_cart = $shopping_cart;
        $this->items = $shopping_cart->getItems();

        // ...
    }

    public function executeAdd()
    {
        // ...

        if ($this->hasRequestParameter('id'))
        {
            $product =
ProductPeer::retrieveByPk($this->getRequestParameter('id'));
            $item = new sfShoppingCartItem('Product',
$this->getRequestParameter('id'));
            $item->setQuantity(1);
            $item->setPrice($product->getPrice());
            $shopping_cart = $this->getUser()->getShoppingCart();
            $shopping_cart->addItem($item);
        }

        // ...
    }

    public function executeUpdate()
    {
        $shopping_cart = $this->getUser()->getShoppingCart();
        foreach ($shopping_cart->getItems() as $item)
        {
            if ($this->hasRequestParameter('quantity_'. $item->getId()))
            {
                $item->setQuantity($this->getRequestParameter('quantity_'. $item->getId()));
            }
        }

        // ...
    }

    public function executeDelete()
    {
        if ($this->hasRequestParameter('id'))
        {
            $shopping_cart = $this->getUser()->getShoppingCart();
            $shopping_cart->deleteItem('Product', $requests->getParameter('id'));
        }
    }
}
```

Listing
1-6

```

    }
    // ...
}
...
}

```

Add an item

Let's take a closer look at this code.

To add an item to the shopping cart, you call the `addItem()` method, passing it a `SfShoppingCartItem` object. This object contains the object class and the unique id of the item to be added, the quantity to be added and the price of the item. This allows the shopping cart to contain objects of any class. For example, you could have a shopping cart containing books and CDs.

The price is stored at this moment to avoid difference of price between the product addition and the checkout if a product price is modified in between in a back-office (or if the cart can be kept between sessions). This also allows to apply price discount according to the amount ordered by the client:

```

Listing 1-7
if ($quantity > 10)
{
    $item->setPrice($product->getPrice() * 0.8);
}
else
{
    $item->setPrice($product->getPrice());
}

```

The problem is that you lose the original price if you apply the discount this way. That's why the `SfShoppingCartItem` object has a `setDiscount()` method that expects a discount percentage:

```

Listing 1-8
if ($quantity > 10)
{
    $item->setPrice($product->getPrice());
    $item->setDiscount(20);
}
else
{
    $item->setPrice($product->getPrice());
}

```

Modify an item

To change the quantity of an item, use the method `setQuantity()` of the `SfShoppingCartItem` object. To delete an item, you can either call the `deleteItem()` method or change the quantity to 0 by calling `setQuantity(0)`.

If a user adds the same item (same class and same id) several times, the shopping cart will increase the quantity of the item and not add a new one:

```

Listing 1-9
$item = new SfShoppingCartItem('Product',
$this->getRequestParameter('id'));
$item->setQuantity(1);

```

```
$item->setPrice($product->getPrice());
$shopping_cart = $this->getUser()->getShoppingCart();
$shopping_cart->addItem($item);
$shopping_cart->addItem($item);

// same as

$item = new sfShoppingCartItem('Product',
$this->getRequestParameter('id'));
$item->setQuantity(2);
$item->setPrice($product->getPrice());
$shopping_cart = $this->getUser()->getShoppingCart();
$shopping_cart->addItem($item);
```

Eventually, you may wonder why the update action uses arguments like 'quantity_2313=4' instead of 'id=2313&quantity=4'. As a matter of fact, the way this action is implemented allows the update of multiple article quantities at one time.

Delete an entire shopping cart

To reset the shopping cart, simply call the `clear()` method of the `sfShoppingCart` instance.

```
$this->getUser()->getShoppingCart()->clear();
```

*Listing
1-10*

Display the Shopping Cart in a Template

The action `shoppingcart/index` should display the content of the shopping cart. Let's examine a possible implementation.

Get the content of the shopping cart

Three methods of the `sfShoppingCart` object will help you get the content of a shopping cart:

- `->getItems()`: array of all the `sfShoppingCartItem` objects in the shopping cart
- `->getItem($class_name, $object_id)`: one specific `sfShoppingCartItem` object
- `->getTotal()`: Total amount of the shopping cart (sum of the quantity*price for each item)

Shopping cart items also have a parameter holder⁵. This means that you can add custom information to any item.

For instance, in a website that sells auto parts, the `sfShoppingCartItem` objects need to store the objects added, but also the vehicle for which the part was bought. This can be simply done by adding:

```
$item->setParameter('vehicle', $vehicle_id);
```

*Listing
1-11*

5. http://www.symfony-project.org/book/1_0/02-Exploring-Symfony-s-Code#chapter_02_sub_parameter_holders



you may need a `getObjects()` method instead of `getItems()`. This method exists but it relies on the Propel⁶ data access layer. As the use of Propel is optional, you might not be able to use it. Learn more about the data access layer in Chapter 8⁷.

Pass the values to the template

In order to display the content of the shopping cart, the `index` action has to define a few variables accessible to the template:

Listing 1-12 // ...

```
$this->shopping_cart = $shopping_cart;
$this->items = $shopping_cart->getItems();
```

The following example shows a simple `indexSuccess.php` template based on an iteration over all the items of the shopping cart to display information about each of them:

Listing 1-13

```
<?php if ($shopping_cart->isEmpty()): ?>

    Your shopping cart is empty.

<?php else: ?>

    <?php foreach ($items as $item): ?>
        <?php $object = call_user_func(array($item->getClass(). 'Peer',
        'retrieveByPK'), $item->getId()) ?>
        <?php echo $object->getLabel() ?><br />
        <?php echo $item->getQuantity() ?><br />
        <?php echo currency_format($item->getPrice(), 'EUR' ) ?>
        <?php if ($item->getDiscount(): >
            (- <?php echo $item->getDiscount() ?> %)
        <?php endif; ?><br />
    <?php endforeach; ?>

    Total : <?php echo currency_format($shopping_cart->getTotal(), 'EUR' )
    ?><br />

<?php endif; ?>
```

Note that this example uses the Propel data access layer. If your project uses another data access layer, this example might need adaptations.

With or without Taxes

By default, all the operations (addition with `$shopping_cart->addItem()`, access with `$item->getPrice()` and `$shopping_cart->getTotal()` use prices **without taxes**.

To get the total amount with taxes, you have to call:

Listing 1-14 `$total_with_taxes = $shopping_cart->getTotalWithTaxes()`

6. <http://propel.phpdb.org/trac/>

7. http://www.symfony-project.org/book/1_0/08-Inside-the-Model-Layer

If you need it, the `sfShoppingCart` object can be initialized so that the `add` and `get` methods use the price including taxes:

```
class myUser extends sfUser
{
    public function getShoppingCart()
    {
        if (!$this->hasAttribute('shopping_cart'))
        {
            $this->setAttribute('shopping_cart', new
sfShoppingCart(sfConfig::get('app_cart_tax')));
        }

        $this->getAttribute('shopping_cart')->setUnitPriceWithTaxes(sfConfig::get('app_cart_wit

            return $this->getAttribute('shopping_cart');
        }

        // ...
    }
}
```

*Listing
1-15*

If `sfConfig::get('app_cart_withtaxes')` is set to `true`, the `$shopping_cart->addItem()` and `$item->getPrice()` methods will use prices with taxes. The `getTotal()` and `getTotalWithTaxes()` methods still give the correct results.

Once again, it is a good habit to keep the tax configuration in a configuration file: that's why the example above uses `sfConfig::get('app_cart_withtaxes')` instead of a simple `true`. The `myproject/apps/myapp/config/app.yml` should contain:

```
all:
  cart:
    tax:      19.6
    withtaxes: true
```

*Listing
1-16*

If you are unsure of the way taxes are handled, just ask the shopping cart:

```
$uses_tax = $shopping_cart->getUnitPriceWithTaxes();
```

*Listing
1-17*

