



# The symfony Cookbook

symfony 1.2

This PDF is brought to you by  
**SENSIOLABS** 

*License:* Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported License  
*Version:* cookbook-1.2-en-2010-05-24

# Table of Contents

<b>How to make sortable lists.....</b>	<b>3</b>
Overview .....	3
What you need .....	3
Data structure .....	3
Extending the model .....	4
Preparing the module.....	6
Classic sortable list.....	6
AJAX sortable list .....	8
Base .....	8
Handling the AJAX request.....	9
Focus on the <code>sortable_element()</code> options .....	10
Comparison.....	11

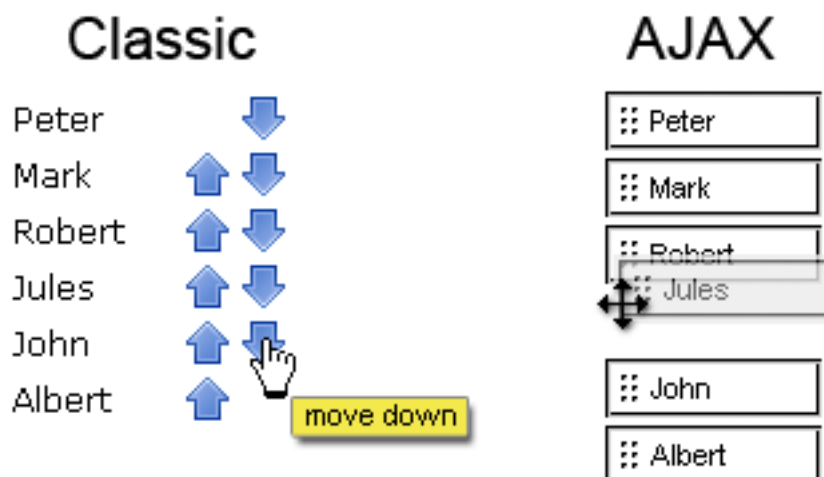
## Chapter 1

# How to make sortable lists

## Overview

Many web applications need to offer an interface to order items - think about categories in a weblog, articles in a CMS, wishes in an e-commerce website... The old fashion way of doing it is to offer arrows to move one item up or down in the list. The AJAX way of doing it is to allow direct drag-and-drop ordering with server support. This chapter will describe both ways, together with a few tips on the way to enhance your object model and to do complex queries with Creole.

## Ordered list of items



## What you need

### Data structure

For this article, the example used will be an undefined `Item` table - name it according to your needs. In order to be sortable, records need at least a `rank` field - no need for a `heap`<sup>1</sup> here

---

1. [http://en.wikipedia.org/wiki/Heap %28data structure%29](http://en.wikipedia.org/wiki/Heap_%28data_structure%29)

since the sorting will be done by the user, not by the computer. So the data structure (to be written in the `schema.yml`<sup>2</sup>) is simply:

```
Listing 1-1 propel:
  test_item:
    _attributes: { phpName: Item }
    id:
    name:         varchar(255)
    rank:         { type: integer, required: true }
```

Make sure you build your model once the data structure is defined by typing in a command line interface:

```
Listing 1-2 $ symfony propel-build-model
```

You will also need a database with the same structure. The fastest way of doing it is to call:

```
Listing 1-3 $ symfony propel-build-sql
$ symfony propel-insert-sql
```

## Extending the model

Before thinking about the user interactions, make sure you have a way to retrieve items by rank, to get the list of items ordered by rank, and to get the current maximum rank, by adding the following methods to the `lib/model/ItemPeer.php`:

```
Listing 1-4 static function retrieveByRank($rank = 1)
{
    $c = new Criteria;
    $c->add(self::RANK, $rank);
    return self::doSelectOne($c);
}

static function getAllByRank()
{
    $c = new Criteria;
    $c->addAscendingOrderByColumn(self::RANK);
    return self::doSelect($c);
}

static function getMaxRank()
{
    $con = Propel::getConnection(self::DATABASE_NAME);
    $sql = 'SELECT MAX(' . self::RANK . ') AS max FROM ' . self::TABLE_NAME;
    $stmt = $con->prepare($sql);
    $stmt->execute();

    $row = $stmt->fetch();
    return $row['max'];
}
```

These methods will be of great use for both sorting interfaces. If you need more information about the way the Object Model handles database queries in symfony, check out the basic CRUD chapter<sup>3</sup> of the Propel user guide.

---

2. [http://www.symfony-project.org/book/1\\_2/08-Inside-the-Model-Layer#chapter\\_08\\_symfony\\_s\\_database\\_schema](http://www.symfony-project.org/book/1_2/08-Inside-the-Model-Layer#chapter_08_symfony_s_database_schema)

There are two more method that needs to be added to the `lib/model/Item.php` class. They won't be needed here, but you will probably need them in a real world application, where you will also add and delete items to your table:

```
public function save(PropelPDO $con = null)
{
    // New records need to be initialized with rank = maxRank +1
    if(!$this->getId())
    {
        $con = Propel::getConnection(ItemPeer::DATABASE_NAME);
        try
        {
            $con->beginTransaction();

            $this->setRank(ItemPeer::getMaxRank()+1);
            parent::save();

            $con->commit();
        }
        catch (Exception $e)
        {
            $con->rollback();
            throw $e;
        }
    }
    else
    {
        parent::save();
    }
}

public function delete(PropelPDO $con = null)
{
    $con = Propel::getConnection(ItemPeer::DATABASE_NAME);
    try
    {
        $con->beginTransaction();

        // decrease all the ranks of the page records of the same category
        // with higher rank
        $sql = 'UPDATE '.ItemPeer::TABLE_NAME.' SET '.ItemPeer::RANK.' =
        '.ItemPeer::RANK.' - 1 WHERE '.ItemPeer::RANK.' > '.$this->getRank();
        $con->exec($sql);
        // delete the item
        parent::delete();

        $con->commit();
    }
    catch (Exception $e)
    {
        $con->rollback();
        throw $e;
    }
}
```

Listing  
1-5

3. <http://propel.phpdb.org/trac/wiki/Users/Documentation/BasicCRUD>

Additions and deletions of records have to be managed carefully for the integrity of the rank field, that's why the `save()` and `delete()` methods are to be specialized. Because these methods do complex read/write operations and create a risk of concurrency issues, these operations are enclosed in a transaction<sup>4</sup> (refer to the Propel documentation<sup>5</sup> for more details about transactions in symfony).

## Preparing the module

The interactions described in this tutorial will take place in a `item` module. Initialize it by calling (assuming you have a frontend application):

*Listing 1-6* `$ symfony init-module frontend item`

Make sure your web server configuration is OK by testing the access to this new module via your favorite browser. Here is the URL that you should check if you follow this tutorial with a sandbox:

*Listing 1-7* `http://localhost/sf_sandbox/web/frontend_dev.php/item`

Finally, if you want to test the ordering of items, you will need... items. Create a bunch of test items, either via a CRUD interface<sup>6</sup> or a population file<sup>7</sup>.

Now that everything is ready, let's get started.

## Classic sortable list

A classic sortable list is a list for which each item has a control to change its order. First, create the action and template that display the list:

*Listing 1-8*

```
// add to modules/item/actions/actions.class.php
public function executeList()
{
    $this->items = ItemPeer::getAllByRank();
    $this->max_rank = ItemPeer::getMaxRank();
}

// create a template listSuccess.php in modules/item/templates/
<h1>Ordered list of items</h1>
<ul>
<?php foreach($items as $item): ?>
    <li>
        <?php
            echo $item->getName().' ';
            if($item->getRank() > 0):
                echo link_to('Move up ', 'item/up?id='.$item->getId());
            endif;
            if($item->getRank() != $max_rank):
                echo link_to('Move down', 'item/down?id='.$item->getId());
            endif;
        </?php
    </li>
</ul>
```

4. [http://en.wikipedia.org/wiki/Database\\_transaction](http://en.wikipedia.org/wiki/Database_transaction)

5. <http://propel.phpdb.org/trac/wiki/Users/Documentation/HowTos/Transactions>

6. [http://www.symfony-project.org/book/1\\_2/14-Generators#chapter\\_14\\_code\\_generation\\_based\\_on\\_the\\_model](http://www.symfony-project.org/book/1_2/14-Generators#chapter_14_code_generation_based_on_the_model)

7. [http://www.symfony-project.org/book/1\\_2/16-Application-Management-Tools#chapter\\_16\\_populating\\_a\\_database](http://www.symfony-project.org/book/1_2/16-Application-Management-Tools#chapter_16_populating_a_database)

```

    ?>
  </li>
<?php endforeach ?>
</ul>

```

The links to move an item up or down are displayed only when the reordering is possible. This means that the first item cannot be moved further up, and the last item cannot be moved further down. Check that the page displays correctly:

[http://localhost/sf\\_sandbox/web/frontend\\_dev.php/item/list](http://localhost/sf_sandbox/web/frontend_dev.php/item/list)

*Listing  
1-9*

Now, it's time to look into the `item/up` and `item/down` action. The `up` action is supposed to decrease the rank of the page given as a parameter, and to increase the rank of the previous page. The `down` action is supposed to increase the rank of the page given as parameter, and to decrease the rank of the following page. As they both do two write operations in the database, these actions should use a transaction.

The two actions have a very similar logic, and if you want to keep D.R.Y.<sup>8</sup>, you'd better find a smart way to write them without repeating any code. This is done by adding a `swapWith()` method to the `Item.php` model class:

```

public function swapWith($item)
{
    $con = Propel::getConnection(ItemPeer::DATABASE_NAME);
    try
    {
        $con->beginTransaction();

        $rank = $this->getRank();
        $this->setRank($item->getRank());
        $this->save();
        $item->setRank($rank);
        $item->save();

        $con->commit();
    }
    catch (Exception $e)
    {
        $con->rollback();
        throw $e;
    }
}

```

*Listing  
1-10*

Then, the `up` and `down` actions become pretty simple:

```

public function executeUp()
{
    $item = ItemPeer::retrieveByPk($this->getRequestParameter('id'));
    $this->forward404Unless($item);
    $previous_item = ItemPeer::retrieveByRank($item->getRank() - 1);
    $this->forward404Unless($previous_item);
    $item->swapWith($previous_item);

    $this->redirect('item/list');
}

```

*Listing  
1-11*

---

8. <http://c2.com/cgi/wiki?DontRepeatYourself>

```

public function executeDown()
{
    $item = ItemPeer::retrieveByPk($this->getRequestParameter('id'));
    $this->forward404Unless($item);
    $next_item = ItemPeer::retrieveByRank($item->getRank() + 1);
    $this->forward404Unless($next_item);
    $item->swapWith($next_item);

    $this->redirect('item/list');
}

```

If not for the security checks made by the calls to `forward404Unless()`, these actions could be even simpler, but you have to protect your application against wrong requests - the ones that could be done by typing an URL directly.

The list is now fully functional. Try it by moving items up and down in the list.

## AJAX sortable list

### Base

Developing a basic AJAX sortable list is not harder than developing a classic one. Most of the job is handled by a special JavaScript helper called `sortable_element()`:

```

Listing // add to modules/item/actions/actions.class.php
1-12
public function executeAjaxList()
{
    $this->items = ItemPeer::getAllByRank();
}

// create a template ajaxListSuccess.php in modules/item/templates/
<?php use_helper('Javascript') ?>
<style>
    .sortable { cursor: move; }
</style>
<h1>Ordered list of items - AJAX enabled</h1>
<ul id="order">
    <?php foreach($items as $item): ?>
        <li id="item_<?php echo $item->getId() ?>" class="sortable">
            <?php echo $item->getName() ?>
        </li>
    <?php endforeach ?>
</ul>
<div id="feedback"></div>
<?php echo sortable_element('order', array(
    'url' => 'item/sort',
    'update' => 'feedback',
)) ?>

```

Check out the result by typing:

```

Listing http://localhost/sf_sandbox/web/frontend_dev.php/item/ajaxlist
1-13

```

By the magic of the `sortable_element()` JavaScript helper, the `<ul>` element is made sortable, which means that its children can be reordered by drag and drop. Every time that

the user drags an item and releases it to reorder the list, an AJAX request is made with the following parameters:

```
POST /sf_sandbox/web/frontend_dev.php/item/sort HTTP/1.1
order[]=1&order[]=3&order[]=2&order[]=4&order[]=5&order[]=6&_ =
```

*Listing  
1-14*

The full ordered list is passed as an array (with the format `order[$rank]=$id`, the `$order` starting at 0 and the `$id` being based on what comes after the underscore (`_`) in the list element `id` property). The `id` property of the sortable element (`order` in the example) is used to name the array of parameters. The JavaScript helper makes a `XMLHttpRequest` to the `url` action (`item/sort` in the example), passing the ordered list in `POST` mode, and uses the result of the action to update the element of `id` update (the feedback div in the example)

## Handling the AJAX request

Now let us write the `item/sort` action and see how it reorders the list of items:

```
// add to modules/item/actions/actions.class.php
public function executeSort()
{
    $order = $this->getRequestParameter('order');
    $flag = ItemPeer::doSort($order);
    return $flag ? sfView::SUCCESS : sfView::ERROR;
}
```

*Listing  
1-15*

The ability to reorder the whole list is part of the model, that's why it is implemented a static method of the `ItemPeer` class. Once again, the fact that this method updates many records of the `item` table makes it necessary to enclose the updates in a database transaction.

```
static function doSort($order)
{
    $con = Propel::getConnection(self::DATABASE_NAME);
    try
    {
        $con->beginTransaction();

        foreach ($order as $rank => $id)
        {
            $item = ItemPeer::retrieveByPk($id);
            if($item->getRank() != $rank)
            {
                $item->setRank($rank);
                $item->save();
            }
        }

        $con->commit();
        return true;
    }
    catch (Exception $e)
    {
        $con->rollback();
        return false;
    }
}
```

*Listing  
1-16*

The value returned by this method will determine which template the action will display. Add the following templates in your `modules/item/templates/` folder:

```
Listing // sortSuccess.php
1-17 Ok

// sortError.php
<strong>A problem occurred. Please refresh and try again.</strong>
```

Test the server handling by pressing F5 after rearranging the list. The ordering shouldn't change, proving that the server understood and saved correctly what the AJAX request sent to it.

## Focus on the `sortable_element()` options

The Javascript helpers chapter<sup>9</sup> describes the generic options of remote function calls, but this example is a good opportunity to see the ones of the `sortable_element()` in detail.

You can define a **different appearance for hovered list elements** when dragging another element over them with the `hoverclass` parameter:

```
Listing <?php use_helper('Javascript') ?>
1-18 <style>
    .sortable { cursor: move; }
    .hovered  { font-weight: bold; }
</style>
...
<?php echo sortable_element('order', array(
    'url'      => 'item/sort',
    'hoverclass' => 'hovered',
)) ?>
```

You can **add non-sortable elements** to the list and restrict the drag-and-drop behaviour to a single class only with the `only` parameter:

```
Listing ...
1-19 <ul id="order">
    <?php foreach($items as $item): ?>
    <li id="item_<?php echo $item->getId() ?>" class="sortable">
        <?php echo $item->getName() ?>
    </li>
    <?php endforeach ?>
    <li>This element is not part of the ordered list</li>
</ul>
<?php echo sortable_element('order', array(
    'url'      => 'item/sort',
    'only'    => 'sortable',
)) ?>
```

If the list elements are not displayed vertically like in the previous example, you have to set the `overlap` parameter to `horizontal`:

```
Listing <?php use_helper('Javascript') ?>
1-20 <style>
    .sortable { cursor: move; float: left; }
```

9. [http://www.symfony-project.org/book/1\\_2/11-Ajax-Integration#chapter\\_11\\_remote\\_call\\_parameters](http://www.symfony-project.org/book/1_2/11-Ajax-Integration#chapter_11_remote_call_parameters)

```

</style>
...
<?php echo sortable_element('order', array(
    'url'      => 'item/sort',
    'overlap' => 'horizontal',
)) ?>

```

If the list to order is not a set of `<li>` elements, you will have to define which child elements of the sortable element are to be made draggable:

```

...
<div id="order">
    <?php foreach($items as $item): ?>
        <div id="item_<?php echo $item->getId() ?>" class="sortable">
            <?php echo $item->getName() ?>
        </div>
        <?php endforeach ?>
        <p>This cannot be dragged</p>
    </div>
<?php echo sortable_element('order', array(
    'url'      => 'item/sort',
    'tag'      => 'div',
)) ?>

```

Listing  
1-21

For all AJAX actions, it is good to have a **visual feedback** of background activity and of the success of the request:

```

<div id="feedback"></div>
<div id="indicator" style="display:none;"></div>
<?php echo sortable_element('order', array(
    'url'      => 'item/sort',
    'update'   => 'feedback',
    'loading'  => "Element.show('indicator')",
    'complete' => "Element.hide('indicator')",
    'success'  => visual_effect('highlight', 'feedback'),
)) ?>

```

Listing  
1-22

For more details about these parameters and about some others that are not described here, refer to the [script.aculo.us](http://script.aculo.us) Sortable manual<sup>10</sup>.

## Comparison

The two methods are both effective to sort a list, but there are limitations and drawbacks.

For large arrays of items, you will probably need a paginated list. The classic method works fine with a page-by-page list, but the AJAX one needs adaptations, and makes it impossible to rearrange elements outside of their own page. That's why you should probably provide a 'move item to position' feature in addition to the AJAX ordering interface.

The AJAX action is not as well protected against wrong requests as the classic one. In order to avoid any risk of database incoherence, you should add a `validateSort()` method to the `itemActions` class. This method would check that all the items id, and only them, are present once and only once in the received array.

---

10. <http://wiki.script.aculo.us/scriptaculous/show/Sortable.create>

One drawback of the `ItemPeer::doSort()` method used in the AJAX sorting is the number of queries it needs to reorder the list. Each movement in a list of  $n$  items makes at least  $n+2$  queries to the database. AJAX lists are not adapted to long lists, so this may not be a major problem, but if performance is a concern for you, you should refactor this method to have it update the ranks with only one query - for instance using the `UPDATE table SET CASE/WHEN SQL` syntax.

The AJAX interface is definitely more user friendly, especially for long ordering tasks, since there is no obligation of a server refresh between two operations. But the ability to drag elements is new in web interfaces, and users not used to it might find it surprising. Moreover, if you choose the AJAX interface, you will have to think about the size of the draggable elements (they need to be large enough to be grabbed), their aspect, their freedom of movement... A lot of Human-Computer Interaction issues that wouldn't need solving with the classic method.

AJAX interactions are always a problem if your target population may turn JavaScripts off in their browser. This means that in addition to the design of a JavaScript interface, you should then provide the classic interface as an alternative so that your functionality degrades gracefully.

All in all, the AJAX version really feels and looks better, but it is at least twice as long to develop.



