# django-admin-sortable2

***Release 2.1.8***

**Jul 04, 2023**

# Contents

is a generic drag-and-drop ordering package to sort objects in the list- and detail inline-views of the Django admin interface. This package offers simple mixin classes which enriches the functionality of *any* existing class derived from `admin.ModelAdmin`, `admin.StackedInline` or `admin.TabularInline`. It thus makes it very easy to integrate with existing models and their model admin interfaces.

Project home: https://github.com/jrief/django-admin-sortable2

CHAPTER 1

# Features

## 1.1 Must not inherit from any special Model base class

Other plugins offering functionality to make list views for the Django admin interface sortable, offer a base class to be used instead of `models.Model`. This class then contains a hard coded position field, additional methods, and meta directives.

By using a mixin to enrich an existing class with sorting, we can integrate this Django-app into existing projects with minimal modification to the code base.

## 1.2 Intuitive List View

By adding a draggable area into one of the columns of the Django admin's list view, sorting rows becomes very intuitive. Alternatively, rows can be selected using the checkbox and sorted as a group.

If rows have to be sorted accross pages, they can be selected using the checkbox and moved to any other page using an Admin action.

## 1.3 Support for Stacked- and Tabular Inlines

If a Django admin view uses InlineModelAdmin objects, and the related model provides an ordering field, then those inline models can be sorted in the detail view.

CHAPTER 2

Contents:

## 2.1 Installation

Install **django-admin-sortable2**. The latest stable release is available on PyPI

```
pip install django-admin-sortable2
```

### 2.1.1 Upgrading from version 1

When upgrading from version 1, check for `StackedInline`- and `TabularInline`-classes inheriting from `SortableInlineAdminMixin`. If they do, check the class inheriting from `ModelAdmin` and using this inline-admin class. Since version 2, this class then also has to inherit from `SortableAdminBase` or a class derived of thereof.

### 2.1.2 Configuration

In the project's `settings.py` file add `'adminsortable2'` to the list of `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.admin',
    'django.contrib.staticfiles',
    'django.contrib.messages',
    ...
    'adminsortable2',
    ...
]
```

The next step is to adopt the models to make them sortable. Please check the page *Using Admin Sortable* for details.

## 2.2 Using Admin Sortable

This library tries to not interfere with existing Django models. Instead it adopts any class inheriting from `Model`, provided it offers a field for sorting.

### 2.2.1 Prepare the Model Classes

Each database model which shall be sortable, requires a position value in its model description. Rather than defining a base class, which contains such a positional value in a hard coded field, this library lets you reuse existing sort fields or define a new field for the sort value.

Therefore this module can be applied in situations where your model inherits from an existing abstract model which already contains any kind of position value. The only requirement is, that this position value be specified as the primary field used for sorting. This in Django is declared through the model's `Meta` class. Here's an example `models.py`:

```
class SortableBook(models.Model):
    title = models.CharField(
        "Title",
        max_length=255,
    )

    my_order = models.PositiveIntegerField(
        default=0,
        blank=False,
        null=False,
    )

    class Meta:
        ordering = ['my_order']
```

Here the ordering field is named `my_order`, but any valid Python variable name will work. There are some constraints though:

- `my_order` is the first field in the `ordering` list (or tuple) of the model's `Meta` class. Alternatively the ordering can be specified inside the class inheriting from `ModelAdmin` registered for this this model.
- `my_order` shall be indexed for performance reasons, so add the attribute `db_index=True` to the field's definition.
- `my_order`'s default value must be 0. The JavaScript which performs the sorting is 1-indexed, so this will not interfere with the order of the items, even if they are already using 0-indexed ordering fields.
- The `my_order` field must be editable, so make sure that it **does not** contain attribute `editable=False`.

The field used to store the ordering position may be any kind of numeric model field offered by Django. Use one of these models fields:

- `BigIntegerField`
- `IntegerField`
- `PositiveIntegerField` (recommended)
- `PositiveSmallIntegerField` (recommended for small sets)
- `SmallIntegerField`

In addition to the reccomended fields, `DecimalField` or `FloatField` may work, but haven't been testest.

**Warning:** Do not make this field unique! See below why.

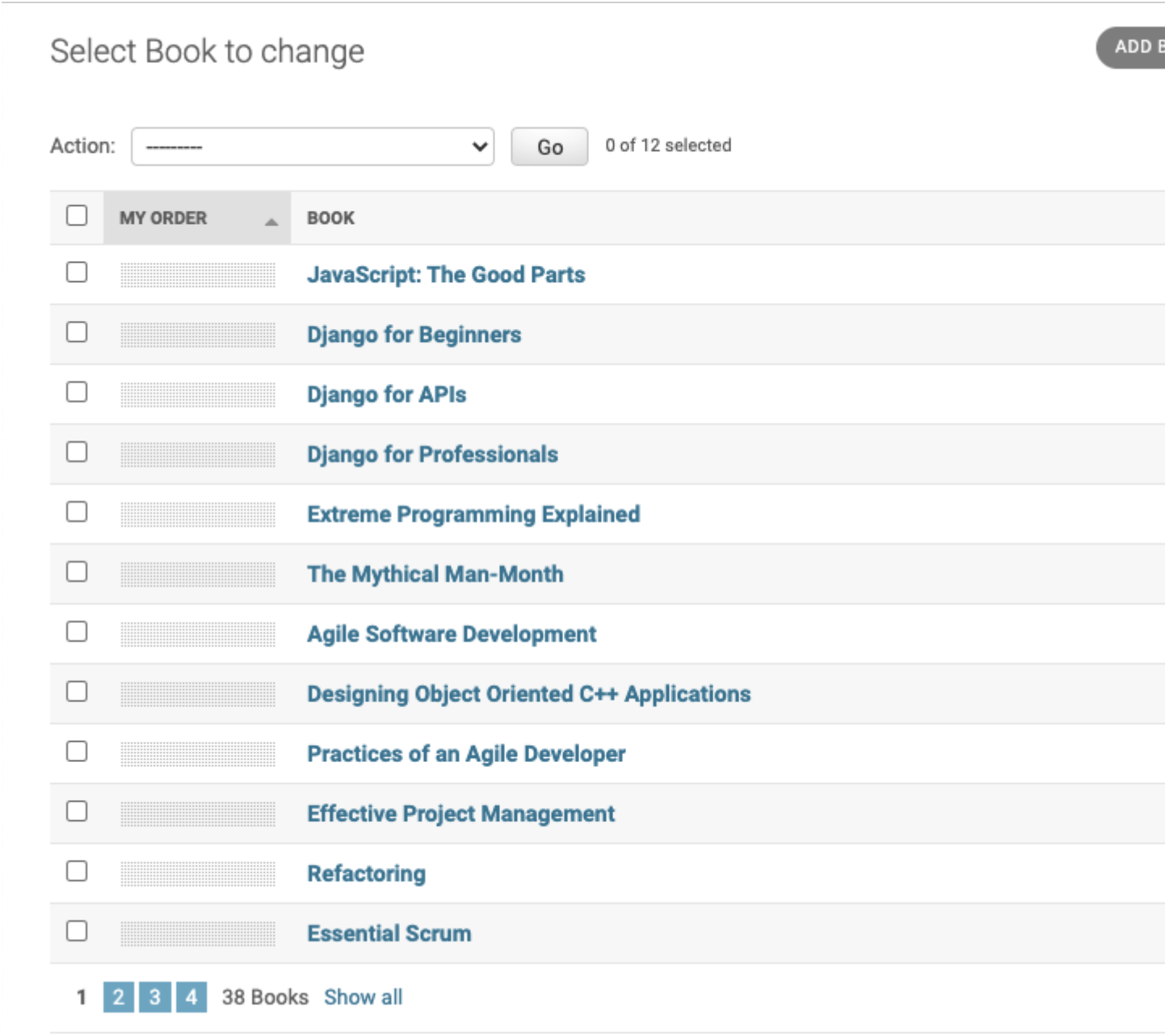
### 2.2.2 In Django's Admin, make the List View sortable

If a models contains an ordering field, all we need to make the Django's Admin List View sortable, is adding the mixin class `adminsortable2.admin.SortableAdminMixin`. By ineriting from this mixin class together with `ModelAdmin`, we get a list view which without any further configuration, owns the functionality to sort its items.

Using the above `SortableBook` model, we can register a default admin interface using

```
from django.contrib import admin
from adminsortable2.admin import SortableAdminMixin

from myapp.models import SortableBook

@admin.register(SortableBook)
class SortableBookAdmin(SortableAdminMixin, admin.ModelAdmin):
    pass
```

This creates a list view with a drag area for each item. By dragging and dropping those items, one can resort the items in the database.




It also is possible to move more than one item at a time. Simply select them using the action checkboxes on the left hand side and move all selected row to a new position.

If the list view is subdivided into more than one page, and items shall be moved to another page, simply select them using the action checkboxes on the left hand side and from the pull down menu named **Action**, choose onto which page the selected items shall be moved.

---

**Note:** In the list view, the ordering field is updated immediatly inside the database.

---

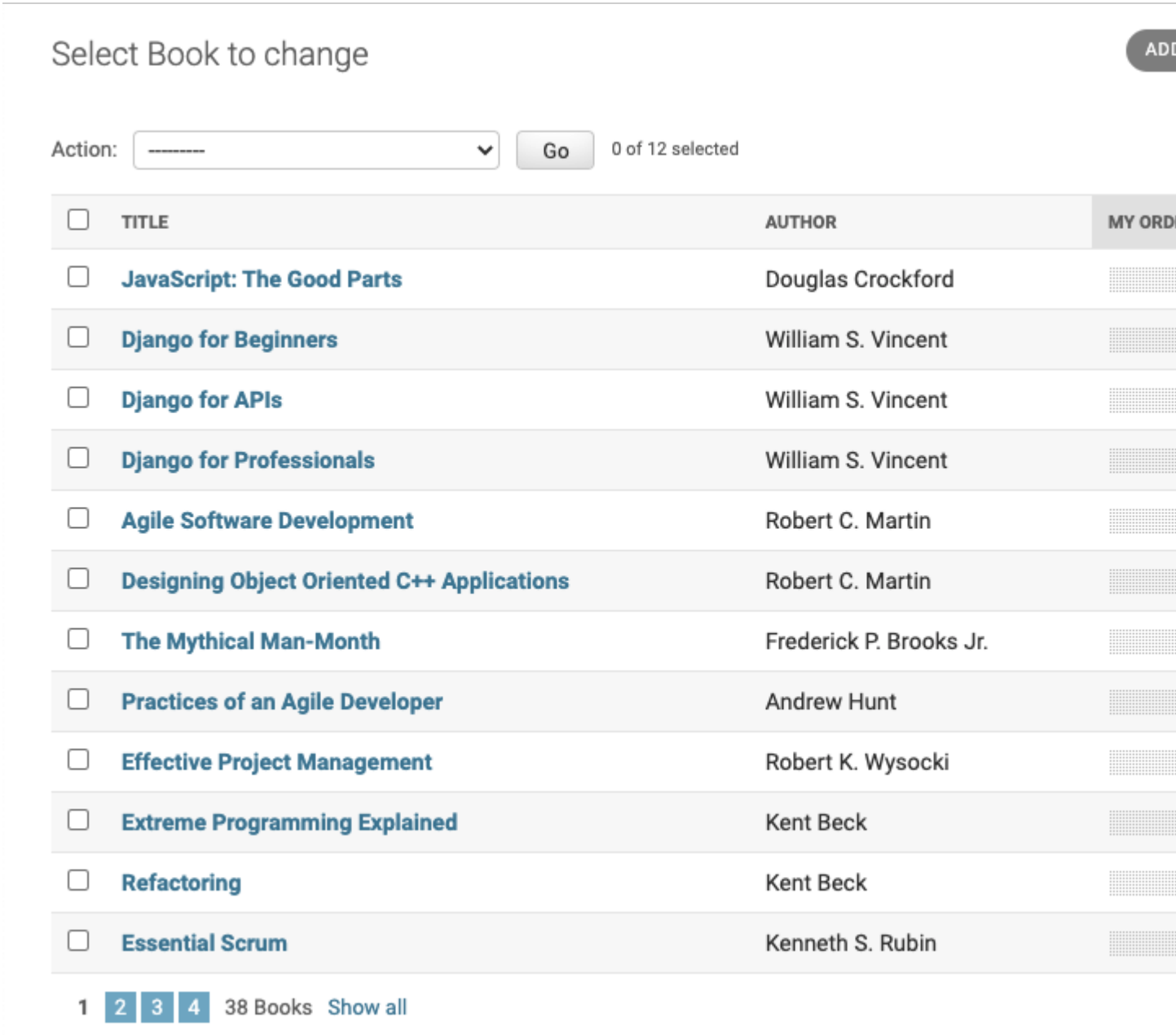
In case the model does not specify a default ordering field in its `Meta` class, it also is possible to specify that field inside the `ModelAdmin` class. The above definition then can be rewritten as:

```
@admin.register(SortableBook)
class SortableBookAdmin(SortableAdminMixin, admin.ModelAdmin):
    ordering = ['my_order']
```

By default, the draggable area is positioned on the first column. If it shall be placed somewhere else, add `list_display` to `SortableBookAdmin` containing the field names of the columns to be rendered in the model's list view. Redefining the above class as:

```
@admin.register(SortableBook)
class SortableBookAdmin(SortableAdminMixin, admin.ModelAdmin):
        list_display = ['title', 'author', 'my_order']
```

will render the list view as:



### 2.2.3 In Django's Admin Detail View, make Stacked- and Tabular-Inlines sortable

If a model on the same page has a parent model, these are called inlines. Suppose we have a sortable model for chapters and want to edit the chapter together with the book's title using the same editor, then Django admin offers the classes `StackedInline` and `TabularInline`. To make these inline admin interfaces sortable, we simple use the mixin class `adminsortable2.admin.SortableAdminMixin`.

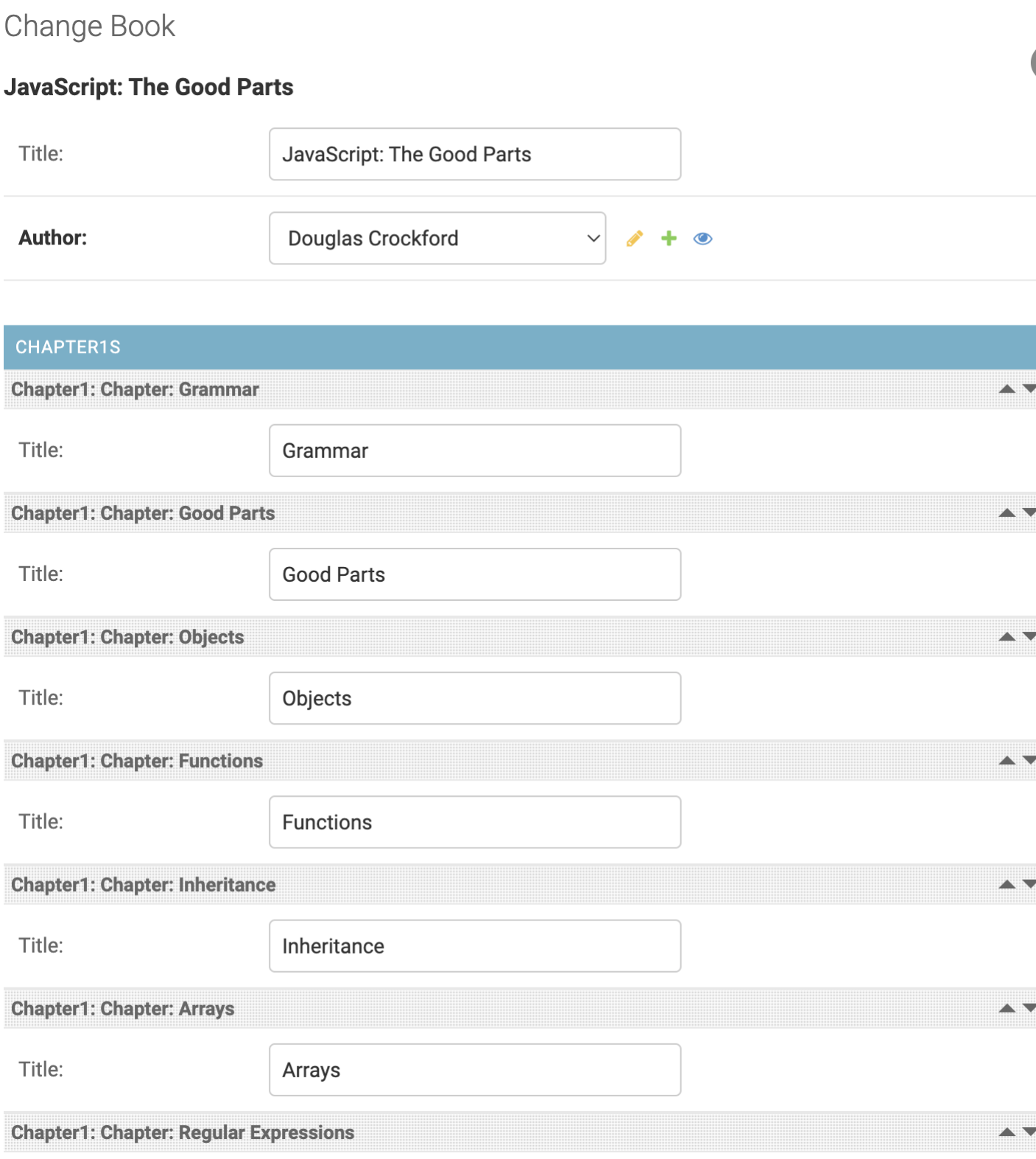
Example:

```
...
from adminsortable2.admin import SortableStackedInline

from myapp.models import Chapter

class ChapterStackedInline(SortableStackedInline):
    model = Chapter

@admin.register(SortableBook)
class SortableBookAdmin(SortableAdminMixin, admin.ModelAdmin):
    ...
    inlines = [ChapterStackedInline]
```

In case model `Chapter` shall be sortable, but model `Book` doesn't have to, rewrite the above class as:

```
...
from adminsortable2.admin import SortableAdminBase

@admin.register(Book)
class SortableBookAdmin(SortableAdminBase, admin.ModelAdmin):
    ...
    inlines = [ChapterStackedInline]
```

For stacked inlines, the editor for the book's detail view looks like:

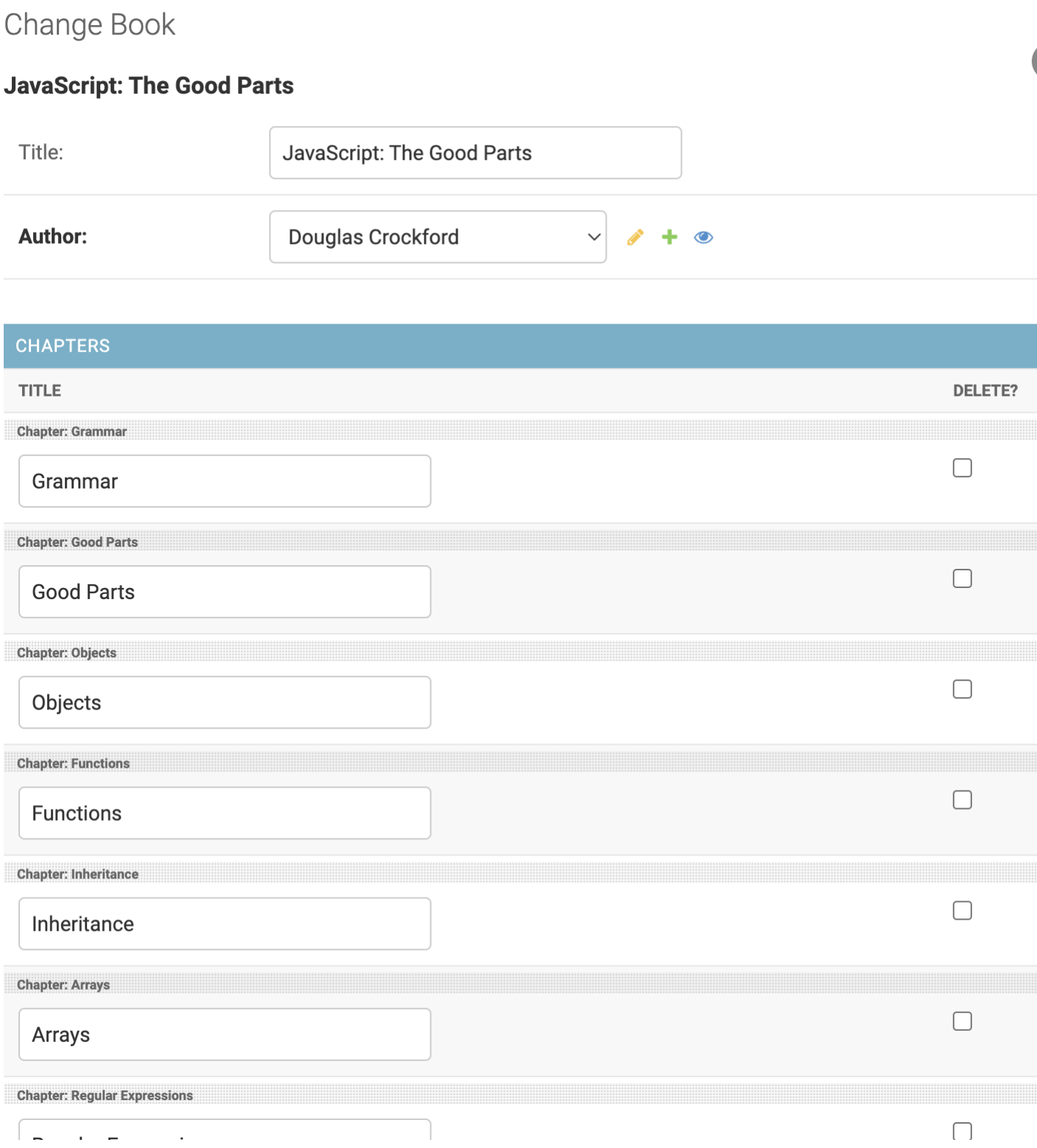



**Note:** Since version 2.1, two buttons have been added to the draggable area above each inline form. They serve to move that edited item to the begin or end of the list of inlines.

If we instead want to use the tabluar inline class, then modify the code from above to

```
...
from adminsortable2.admin import SortableTabularInline

from myapp.models import Chapter

class ChapterTabularInline(SortableTabularInline):
    model = Chapter

@admin.register(SortableBook)
class SortableBookAdmin(SortableAdminMixin, admin.ModelAdmin):
    ...
    inlines = [ChapterTabularInline]
```

the editor for the book's detail view then looks like:




**Note:** When sorting items in the stacked or tabular inline view, these changes are not updated immediatly inside the database. Instead the parent model must explicitly be saved.

### 2.2.4 Sortable Many-to-Many Relations with Sortable Inlines

Sortable many to many relations can be achieved by creating a model to act as a juction table and adding an ordering field. This model can be specified on the `models.ManyToManyField through` parameter that tells the Django ORM to use your juction table instead of creating a default one. Otherwise, the process is conceptually similar to the above examples.

For example if you wished to have buttons added to control panel able to be sorted into order via the Django Admin interface you could do the following. A key feature of this approach is the ability for the same button to be used on more than one panel.

#### Specify a junction model and assign it to the ManyToManyField

`models.py`

```
from django.db import models

class Button(models.Model):
    """A button"""
    name = models.CharField(max_length=64)
    button_text = models.CharField(max_length=64)

class Panel(models.Model):
    """A Panel of Buttons - this represents a control panel."""
    name = models.CharField(max_length=64)
    buttons = models.ManyToManyField(Button, through='PanelButtons')

class PanelButtons(models.Model):
    """This is a junction table model that also stores the button order for a panel."
↪"
    panel = models.ForeignKey(Panel)
    button = models.ForeignKey(Button)
    button_order = models.PositiveIntegerField(default=0)

    class Meta:
        ordering = ['button_order']
```

#### Setup the Tabular Inlines to enable Buttons to be sorted in Django Admin

`admin.py`

```
from django.contrib import admin
from adminsortable2.admin import SortableInlineAdminMixin
from models import Panel

class ButtonTabularInline(SortableInlineAdminMixin, admin.TabularInline):
    # We don't use the Button model but rather the juction model specified on Panel.
    model = Panel.buttons.through

@admin.register(Panel)
class PanelAdmin(admin.ModelAdmin)
    inlines = (ButtonTabularInline,)
```

### 2.2.5 Initial data

In case you just changed your model to contain an additional sorting field (e.g. `my_order`), which does not yet contain any values, then you **must** set initial ordering values.

**django-admin-sortable2** is shipped with a management command which can be used to prepopulate the ordering field:

```
shell> ./manage.py reorder my_app.ModelOne [my_app.ModelTwo ...]
```

If you prefer to do a one-time database migration, just after having added the ordering field to the model, then create a datamigration.

```
shell> ./manage.py makemigrations myapp
```

this creates **non** empty migration named something like `migrations/0123_auto_20220331_001.py`.

Edit the file and add a data migration:

```
def reorder(apps, schema_editor):
    MyModel = apps.get_model("myapp", "MyModel")
    for order, item in enumerate(MyModel.objects.all(), 1):
        item.my_order = order
        item.save(update_fields=['my_order'])
```

Now add `migrations.RunPython(reorder)` to the list of operations:

```
class Migration(migrations.Migration):
    operations = [
        ....
        migrations.RunPython(reorder, reverse_code=migrations.RunPython.noop),
    ]
```

then apply the changes to the database using:

```
shell> ./manage.py migrate myapp
```

---

**Note:** If you omit to prepopulate the ordering field with unique values, after adding that field to an existing model, then attempting to reorder items in the admin interface will fail.

---

### 2.2.6 Note on unique indices on the ordering field

From a design consideration, one might be tempted to add a unique index on the ordering field. But in practice this has serious drawbacks:

MySQL has a feature (or bug?) which requires to use the `ORDER BY` clause in bulk updates on unique fields.

SQLite has the same bug which is even worse, because it does neither update all the fields in one transaction, nor does it allow to use the `ORDER BY` clause in bulk updates.

Only PostgreSQL does it "right" in the sense, that it updates all fields in one transaction and afterwards rebuilds the unique index. Here one can not use the `ORDER BY` clause during updates, which from the point of view for SQL semantics, is senseless anyway.

See https://code.djangoproject.com/ticket/20708 for details.

Therefore I strongly advise against setting `unique=True` on the position field, unless you want unportable code, which only works with Postgres databases.

## 2.3 Contributing to the Project

- Please ask question on the discussion board.
- Ideas for new features also shall be discussed on that board as well.
- The issue tracker shall *exclusively* be used to report bugs.
- Except for very small fixes (typos etc.), do not open a pull request without an issue.

### 2.3.1 Writing Code

Before hacking into the code, adopt your IDE to respect the projects's .editorconfig file.

When installing from GitHub, you *must* build the JavaScript client using the esbuild TypeScript compiler:

```
git clone https://github.com/jrief/django-admin-sortable2.git
cd django-admin-sortable2
npm install --also=dev
npm run build

# and optionally for a minimized version
npm run minify
```

This then builds and bundles the JavaScript file `adminsortable2/static/adminsortable2/js/adminsortable2.js` which later on is imported by the sortable-admin mixin classes. The minimized version can be imported as `adminsortable2/static/adminsortable2/js/adminsortable2.min.js`

### 2.3.2 Run the Demo App

**django-admin-sotable2** is shipped with a demo app, which shall be used as a reference when reporting bugs, proposing new features or to just get a quick first impression of this library.

Follow these steps to run this demo app. Note that in addition to Python, you also need a recent version of NodeJS.

```
git clone https://github.com/jrief/django-admin-sortable2.git
cd django-admin-sortable2
npm install --include=dev
npm run build
npm run minify
python -m pip install Django
python -m pip install -r testapp/requirements.txt

# we use the default template files and patch them, rather than using our own␣
↪modified one
django_version=$(python -c 'from django import VERSION; print("{0}.{1}".
↪format(*VERSION))')
curl --silent --output adminsortable2/templates/adminsortable2/edit_inline/stacked-
↪django-$django_version.html https://raw.githubusercontent.com/django/django/stable/
↪$django_version.x/django/contrib/admin/templates/admin/edit_inline/stacked.html
curl --silent --output adminsortable2/templates/adminsortable2/edit_inline/tabular-
↪django-$django_version.html https://raw.githubusercontent.com/django/django/stable/
↪$django_version.x/django/contrib/admin/templates/admin/edit_inline/tabular.html
```

(continues on next page)

(continued from previous page)

```
patch -p0 adminsortable2/templates/adminsortable2/edit_inline/stacked-django-$django_
↪version.html patches/stacked-django-4.0.patch
patch -p0 adminsortable2/templates/adminsortable2/edit_inline/tabular-django-$django_
↪version.html patches/tabular-django-4.0.patch

cd testapp
./manage.py migrate
./manage.py loaddata fixtures/data.json
./manage.py runserver
```

Point a browser onto http://localhost:8000/admin/, and go to **Testapp > Books**. There you can test the full set of features available in this Django app.

In section **TESTAPP** there are eight entires named "Book". They all manage the same database model (ie. `Book`) and only differ in the way their sorting is organized: Somtimes by the Django model, somtimes by the Django admin class and in both sorting directions.

### 2.3.3 Reporting Bugs

For me it often is very difficult to comprehend why this library does not work with *your* project. Therefore wheneever you want to report a bug, **report it in a way so that I can reproduce it**.

**Checkout the code, build the client and run the demo** as decribed in the previous section. Every feature offered by **django-admin-sortable2** is implemented in the demo named `testapp`. If you can reproduce the bug there, report it. Otherwise check why your application behaves differently.

### 2.3.4 Running Tests

In version 2.0, many unit tests have been replaced by end-to-end tests using Playwright-Python. In addition, the Django test runner has been replaced by pytest-django.

Follow these steps to run all unit- and end-to-end tests.

```
git clone https://github.com/jrief/django-admin-sortable2.git
cd django-admin-sortable2
npm install --also=dev
npm run build
python -m pip install Django
python -m pip install -r testapp/requirements.txt
python -m playwright install
python -m playwright install-deps
python -m pytest testapp
django_version=$(python -c 'from django import VERSION; print("{0}.{1}".
↪format(*VERSION))')
curl --silent --output adminsortable2/templates/adminsortable2/edit_inline/stacked-
↪django-$django_version.html https://raw.githubusercontent.com/django/django/stable/
↪$django_version.x/django/contrib/admin/templates/admin/edit_inline/stacked.html
curl --silent --output adminsortable2/templates/adminsortable2/edit_inline/tabular-
↪django-$django_version.html https://raw.githubusercontent.com/django/django/stable/
↪$django_version.x/django/contrib/admin/templates/admin/edit_inline/tabular.html
patch -p0 adminsortable2/templates/adminsortable2/edit_inline/stacked-django-$django_
↪version.html patches/stacked-django-4.0.patch
patch -p0 adminsortable2/templates/adminsortable2/edit_inline/tabular-django-$django_
↪version.html patches/tabular-django-4.0.patch
```

### 2.3.5 Adding new Features

If you want to add a new feature to **django-admin-sortable2**, please integrate a demo into the testing app (ie. `testapp`). Doing so has two benefits:

I can understand way better what it does and how that new feature works. This increases the chances that such a feature is merged.

You can use that extra code to adopt the test suite.

*Remember*: For UI-centric applications such as this one, where the client- and server-side are strongly entangled with each other, I prefer end-to-end tests way more rather than unit tests. Reason is, that otherwise I would have to mock the interfaces, which itself is error-prone and additional work.

*Don't hide yourself*: I will not accept large pull requests from anonymous users, so please publish an email address in your GitHub's profile. Reason is that when refactoring the code, I must be able to contact the initial author of a feature not added by myself.

### 2.3.6 Quoting

Please follow these rules when quoting strings:

- A string intended to be read by humans shall be quoted using double quotes: *"..."*.
- An internal string, such as dictionary keys, etc. (and thus usually not intended to be read by humans), shall be quoted using single quotes: *'...'*. This makes it easier to determine if we have to extra check for wording.

There is a good reason to follow this rule: Strings intended for humans, sometimes contain apostrophes, for instance *"This is John's profile"*. By using double quotes, those apostrophes must not be escaped. On the other side whenever we write HTML, we have to use double quotes for parameters, for instance *'<a href="https://example.org">Click here!</a>'*. By using single quotes, those double quotes must not be escaped.

### 2.3.7 Lists versus Tuples

Unfortunately in Django, we developers far too often intermixed lists and tuples without being aware of their intention. Therefore please follow this rule:

Always use lists, if there is a theoretical possibility that someday, someone might add another item. Therefore `list_display`, `list_display_links`, `fields`, etc. must always be lists.

Always use tuples, if the number of items is restricted by nature, and there isn't even a theoretical possibility of being extended.

Example:

```
color = ChoiceField(
    label="Color",
    choices=[('ff0000', "Red"), ('00ff00', "Green"), ('0000ff', "Blue")],
)
```

A `ChoiceField` must provide a list of choices. Attribute `choices` must be a list because it is eligible for extension. Its inner items however must be tuples, because they can exlusively containin the choice value and a human readable label. Here we also intermix single with double quotes to distinguish strings intended to be read by the machine versus a human.

CHAPTER 3

# License

CHAPTER 4

# Some Related projects

- https://github.com/jazzband/django-admin-sortable
- https://github.com/mtigas/django-orderable
- https://djangosnippets.org/snippets/2057/
- https://djangosnippets.org/snippets/2306/