# Customizing Change Lists and Forms

Let’s dive into admin customization by specifying the fields that are displayed on the change list for our **Author** model. By default, the change list displays the result of **\_\_str\_\_()** for each object. In Chapter 4, we defined the **\_\_str\_\_()** method for **Author** objects to display the first name and last name together:

**class Author(models.Model):**

**first\_name = models.CharField(max\_length=30)**

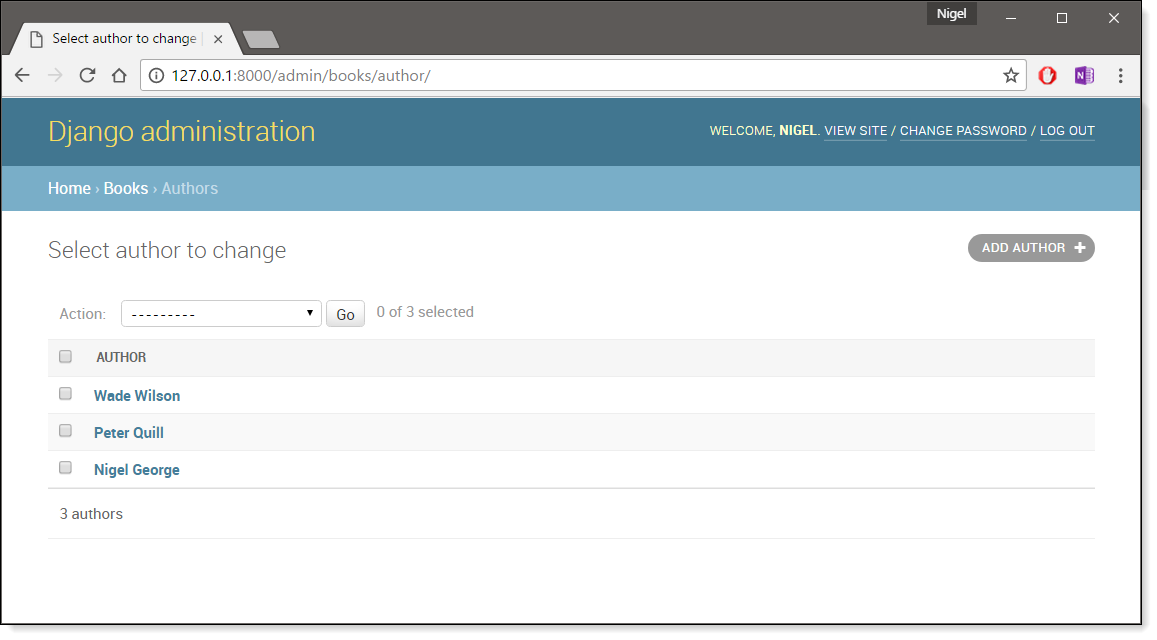
**last\_name = models.CharField(max\_length=40)**

**email = models.EmailField(blank=True, verbose\_name ='e-mail')**

**def \_\_str\_\_(self):**

**return u'%s %s' % (self.first\_name, self.last\_name)**

As a result, the change list for **Author** objects displays each other’s first name and last name together, as you can see in Figure 5-7.

  
**Figure 5-7:** The author change list page

We can improve on this default behavior by adding a few other fields to the change list display. It’d be handy, for example, to see each author’s e-mail address in this list, and it’d be nice to be able to sort by first and last name.

To make this happen, we’ll define a **ModelAdmin** class for the **Author** model. This class is the key to customizing the admin, and one of the most basic things it lets you do is specify the list of fields to display on change list pages. Edit **admin.py** to make these changes:

**from django.contrib import admin**

**from .models import Publisher, Author, Book**

**class AuthorAdmin(admin.ModelAdmin):**

**list\_display = ('first\_name', 'last\_name', 'email')**

**admin.site.register(Publisher)**

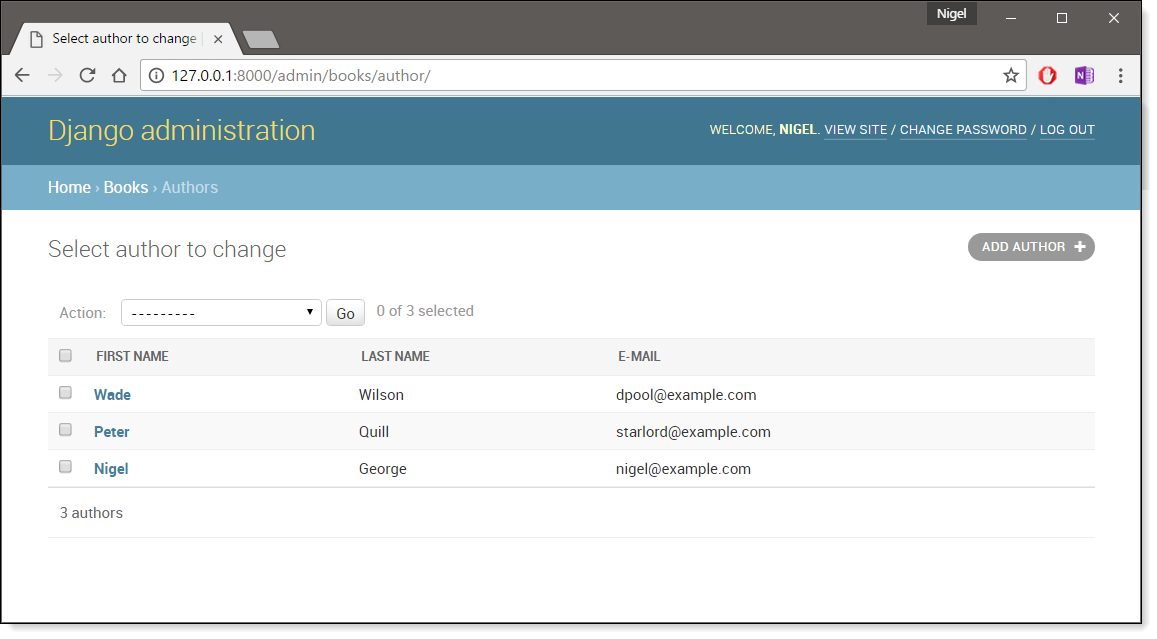
**admin.site.register(Author, AuthorAdmin)**

**admin.site.register(Book)**

Here’s what we’ve done:

* We created the class **AuthorAdmin**. This class, which subclasses **django.contrib.admin.ModelAdmin**, holds custom configuration for a specific admin model. We’ve only specified one customization – **list\_display**, which is set to a tuple of field names to display on the change list page. These field names must exist in the model, of course.
* We altered the **admin.site.register()** call to add **AuthorAdmin** after **Author**. You can read this as: “Register the **Author** model with the **AuthorAdmin** options.” The **admin.site.register()** function takes a **ModelAdmin** subclass as an optional second argument. If you don’t specify a second argument (as is the case for **Publisher** and **Book**), Django will use the default admin options for that model.

With that tweak made, [[restart the server]] reload the author change list page, and you’ll see it’s now displaying three columns – the first name, last name and e-mail address. In addition, each of those columns is sortable by clicking on the column header. (See Figure 5-8.)

  
**Figure 5-8:** The author change list page after list\_display added

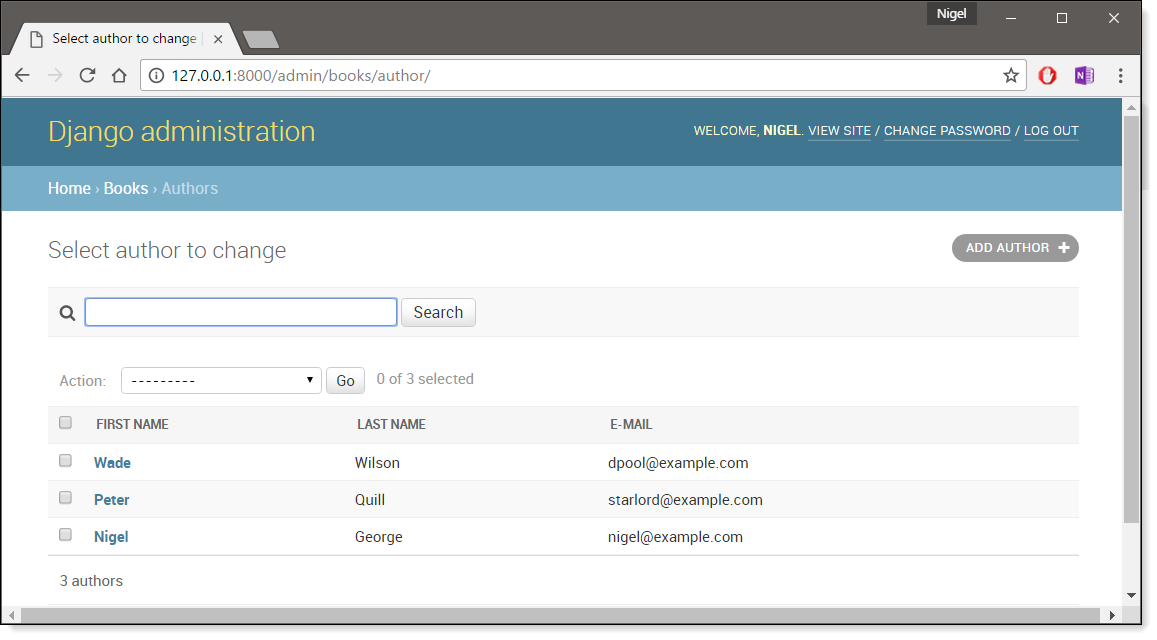
Next, let’s add a simple search bar. Add **search\_fields** to the **AuthorAdmin**, like so:

**class AuthorAdmin(admin.ModelAdmin):**

**list\_display = ('first\_name', 'last\_name', 'email')**

**search\_fields = ('first\_name', 'last\_name')**

Reload the page in your browser, and you should see a search bar at the top. (See Figure 5-9.) We’ve just told the admin change list page to include a search bar that searches against the **first\_name** and **last\_name** fields. As a user might expect, this is case-insensitive and searches both fields, so searching for the string “**bar**” would find both an author with the first name Barney and an author with the last name Hobarson.

  
**Figure 5-9:** The author change list page after search\_fields added

Next, let’s add some date filters to our **Book** model’s change list page:

**from django.contrib import admin**

**from .models import Publisher, Author, Book**

**class AuthorAdmin(admin.ModelAdmin):**

**list\_display = ('first\_name', 'last\_name', 'email')**

**search\_fields = ('first\_name', 'last\_name')**

**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher', 'publication\_date')**

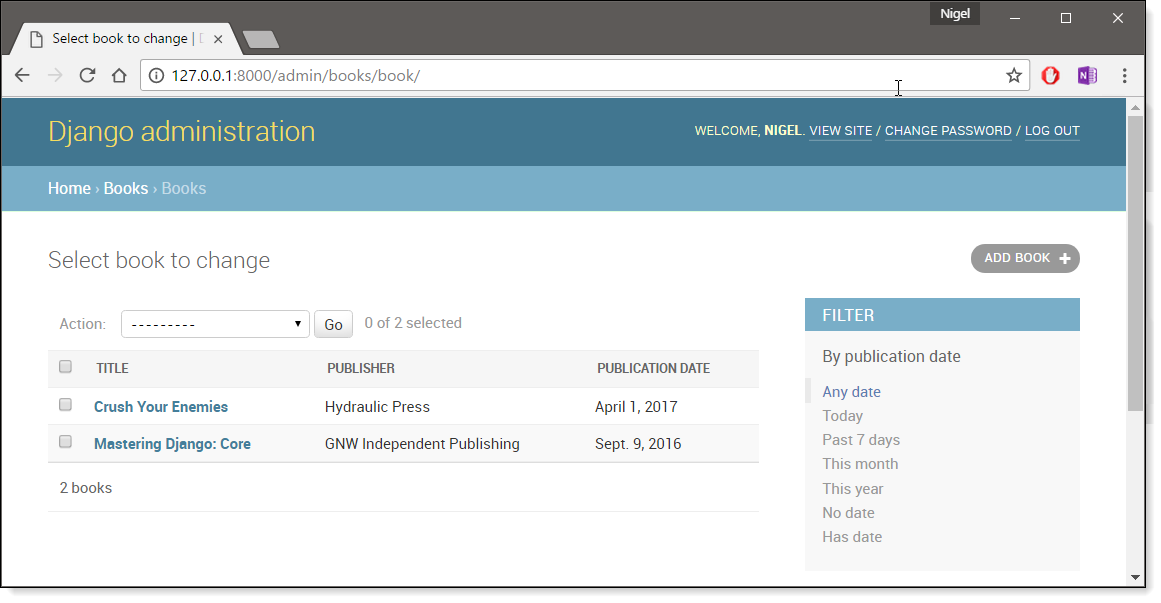
**list\_filter = ('publication\_date',)**

**admin.site.register(Publisher)**

**admin.site.register(Author, AuthorAdmin)**

**admin.site.register(Book, BookAdmin)**

Here, because we’re dealing with a different set of options, we created a separate **ModelAdmin** class – **BookAdmin**. First, we defined a **list\_display** just to make the change list look a bit nicer. Then, we used **list\_filter**, which is set to a tuple of fields to use to create filters along the right side of the change list page. For date fields, Django provides shortcuts to filter the list to “Today,” “Past 7 days,” “This month” and “This year” – shortcuts that Django’s developers have found hit the common cases for filtering by date. Figure 5-10 shows what that looks like.

  
**Figure 5-10:** The book change list page after list\_filter

**list\_filter** also works on fields of other types, not just **DateField**. (Try it with **BooleanField** and **ForeignKey** fields, for example.) The filters show up as long as there are at least 2 values to choose from. Another way to offer date filters is to use the **date\_hierarchy** admin option, like this:

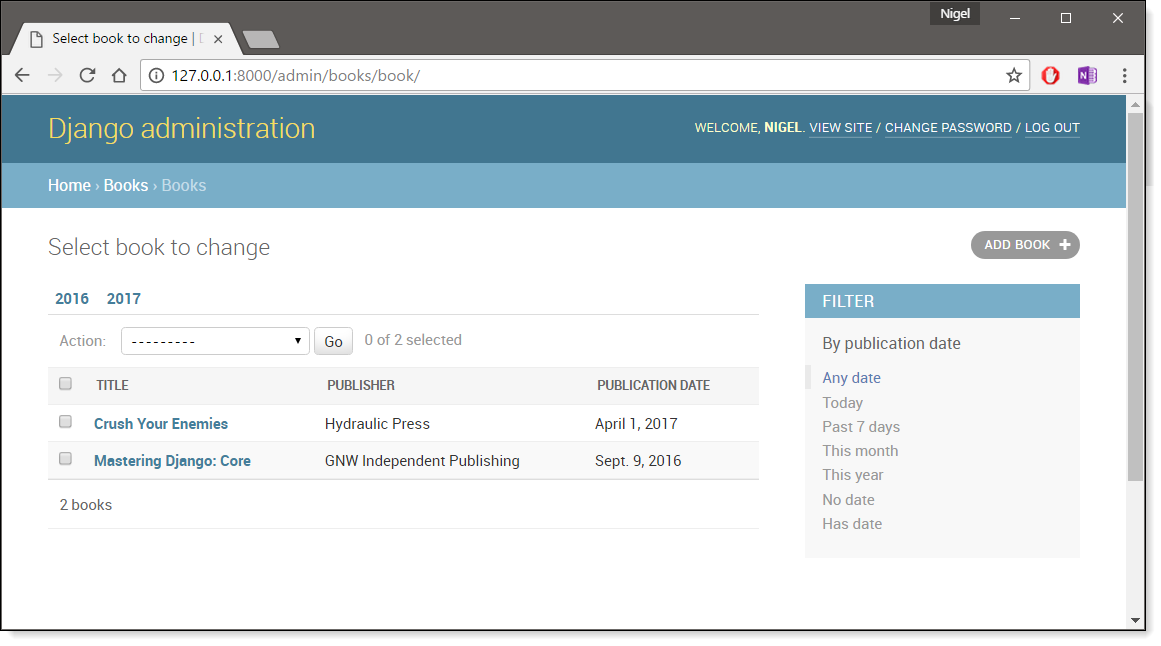
**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher','publication\_date')**

**list\_filter = ('publication\_date',)**

**date\_hierarchy = 'publication\_date'**

With this in place, the change list page gets a date drill-down navigation bar at the top of the list, as shown in Figure 5-11. It starts with a list of available years, then drills down into months and individual days.

  
**Figure 5-11:** The book change list page after date\_hierarchy

Note that **date\_hierarchy** takes a string, not a tuple, because only one date field can be used to make the hierarchy. Finally, let’s change the default ordering so that books on the change list page are always ordered descending by their publication date.

By default, the change list orders objects according to their model’s **ordering** within **class Meta**(which we covered in Chapter 4) – but you haven’t specified this **ordering** value, then the ordering is undefined.

**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher','publication\_date')**

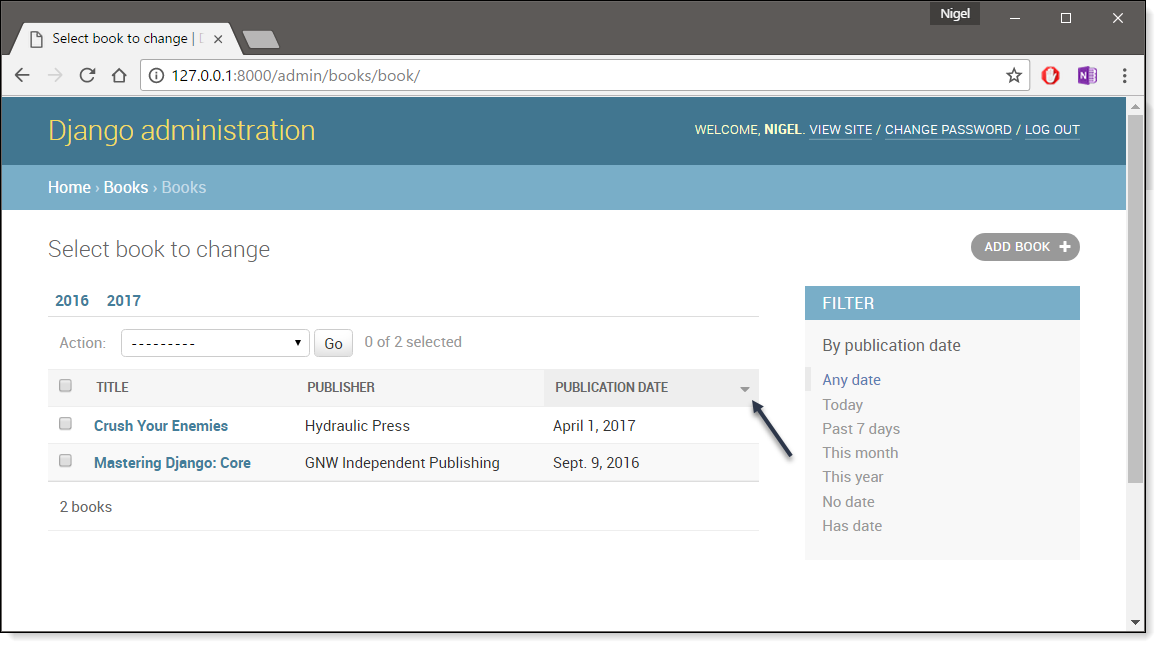
**list\_filter = ('publication\_date',)**

**date\_hierarchy = 'publication\_date'**

**ordering = ('-publication\_date',)**

This admin **ordering** option works exactly as the **ordering** in models’ **class Meta**, except that it only uses the first field name in the list. Just pass a list or tuple of field names, and add a minus sign to a field to use descending sort order.

Reload the book change list to see this in action. Note that the “Publication date” header now includes a small arrow that indicates which way the records are sorted. (See Figure 5-12.)

  
**Figure 5-12:** The book change list page after ordering

We’ve covered the main change list options here. Using these options, you can make a very powerful, production-ready data-editing interface with only a few lines of code.

### Customizing Edit Forms

Just as the change list can be customized, edit forms can be customized in many ways. First, let’s customize the way fields are ordered. By default, the order of fields in an edit form corresponds to the order they’re defined in the model. We can change that using the **fields** option in our **ModelAdmin**subclass:

**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher', 'publication\_date')**

**list\_filter = ('publication\_date',)**

**date\_hierarchy = 'publication\_date'**

**ordering = ('-publication\_date',)**

**fields = ('title', 'authors', 'publisher', 'publication\_date')**

After this change, the edit form for books will use the given ordering for fields.

Another useful thing the **fields** option lets you do is to exclude certain fields from being edited entirely. Just leave out the field(s) you want to exclude.

For example, in our book database, we could hide the **publication\_date** field from being editable:

**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher','publication\_date')**

**list\_filter = ('publication\_date',)**

**date\_hierarchy = 'publication\_date'**

**ordering = ('-publication\_date',)**

**fields = ('title', 'authors', 'publisher')**

As a result, the edit form for books doesn’t offer a way to specify the publication date. This could be useful, say, if you’re an editor who prefers that his authors not push back publication dates. (This is purely a hypothetical example, of course.) When a user uses this incomplete form to add a new book, Django will simply set the **publication\_date** to **None** – so make sure that field has **null=True**.

Another commonly used edit-form customization has to do with many-to-many fields. As we’ve seen on the edit form for books, the admin site represents each **ManyToManyField** as a multiple-select box, which is the most logical HTML input widget to use – but multiple-select boxes can be difficult to use. If you want to select multiple items, you have to hold down the control key, or command on a Mac, to do so.

The admin site helpfully inserts a bit of text that explains this, but it still gets unwieldy when your field contains hundreds of options. The admin site’s solution is **filter\_horizontal**. Let’s add that to **BookAdmin** and see what it does.

**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher','publication\_date')**

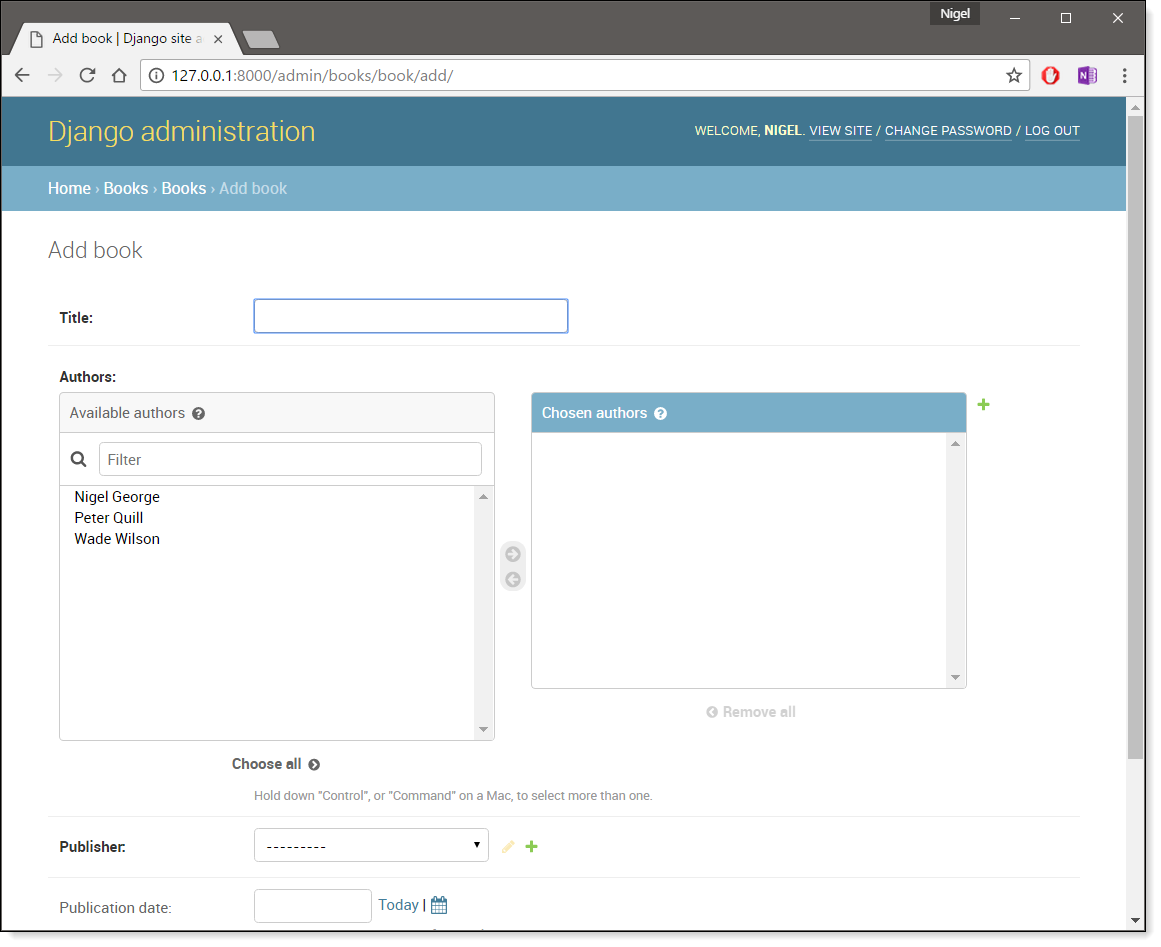
**list\_filter = ('publication\_date',)**

**date\_hierarchy = 'publication\_date'**

**ordering = ('-publication\_date',)**

**filter\_horizontal = ('authors',)**

(If you’re following along, note that I’ve also removed the **fields** option to display all the fields in the edit form.) Reload the edit form for books, and you’ll see that the “Authors” section now uses a fancy JavaScript filter interface that lets you search through the options dynamically and move specific authors from “Available authors” to the “Chosen authors” box, and vice versa (Figure 5-13).

  
**Figure 5-13:** The book edit form after adding filter\_horizontal

I’d highly recommend using **filter\_horizontal** for any **ManyToManyField** that has more than 10 items. It’s far easier to use than a simple multiple-select widget. Also, note you can use **filter\_horizontal**for multiple fields – just specify each name in the tuple.

**ModelAdmin** classes also support a **filter\_vertical** option. This works exactly as **filter\_horizontal**, but the resulting JavaScript interface stacks the two boxes vertically instead of horizontally. It’s a matter of personal taste.

**filter\_horizontal** and **filter\_vertical** only work on **ManyToManyField** fields, not **ForeignKey** fields. By default, the admin site uses simple **<select>** boxes for **ForeignKey** fields, but, as for **ManyToManyField**, sometimes you don’t want to incur the overhead of having to select all the related objects to display in the drop-down.

For example, if our book database grows to include thousands of publishers, the “Add book” form could take a while to load, because it would have to load every publisher for display in the **<select>**box. The way to fix this is to use an option called **raw\_id\_fields**:

**class BookAdmin(admin.ModelAdmin):**

**list\_display = ('title', 'publisher','publication\_date')**

**list\_filter = ('publication\_date',)**

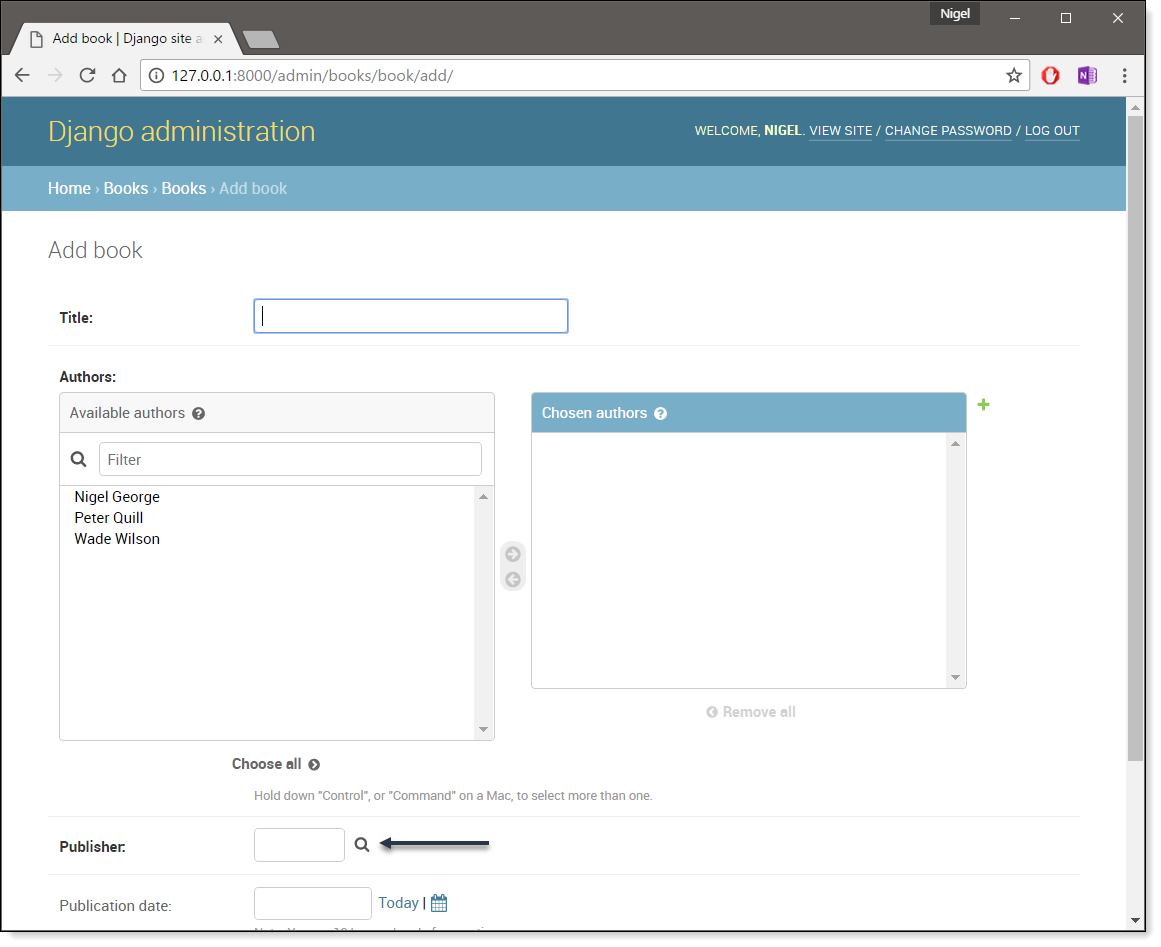
**date\_hierarchy = 'publication\_date'**

**ordering = ('-publication\_date',)**

**filter\_horizontal = ('authors',)**

**raw\_id\_fields = ('publisher',)**

Set this to a tuple of **ForeignKey** field names, and those fields will be displayed in the admin with a simple text input box (**<input type="text">**) instead of a **<select>**. See Figure 5-14.

  
**Figure 5-14:** The book edit form after adding raw\_id\_fields

What do you enter in this input box? The database ID of the publisher. Given that humans don’t normally memorize database IDs, there’s also a magnifying-glass icon that you can click to pull up a pop-up window, from which you can select the publisher to add.