# Writing your first Django app, part 2

This tutorial begins where *Tutorial 1* left off. We're continuing the Web-poll application and will focus on Django's automatically-generated admin site.

### *Admonition*

### *Philosophy*

Generating admin sites for your staff or clients to add, change and delete content is tedious work that doesn't require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.

Django was written in a newsroom environment, with a very clear separation between "content publishers" and the "public" site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

The admin isn't necessarily intended to be used by site visitors; it's for site managers.

## Activate the admin site

The Django admin site is not activated by default -- it's an opt-in thing. To activate the admin site for your installation, do these three things:

- Add `"django.contrib.admin"` to your `INSTALLED_APPS` setting.
- Run `python manage.py syncdb`. Since you have added a new application to `INSTALLED_APPS`, the database tables need to be updated.
- Edit your `mysite/urls.py` file and uncomment the lines below the "Uncomment this for admin:" comments. This file is a URLconf; we'll dig into URLconfs in the next tutorial. For now, all you need to know is that it maps URL roots to applications. In the end, you should have a `urls.py` file that looks like this:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),
)
```

(The bold lines are the ones that needed to be uncommented.)

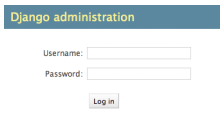
## Start the development server

Let's start the development server and explore the admin site.

Recall from Tutorial 1 that you start the development server like so:

```
python manage.py runserver
```

Now, open a Web browser and go to "/admin/" on your local domain -- e.g., http://127.0.0.1:8000/admin/. You should see the admin's login screen:



# Enter the admin site

Now, try logging in. (You created a superuser account in the first part of this tutorial, remember?) You should see the Django admin index page:

You should see a few other types of editable content, including groups, users and sites. These are core features Django ships with by default.

# Make the poll app modifiable in the admin

But where's our poll app? It's not displayed on the admin index page.

Just one thing to do: We need to tell the admin that `Poll` objects have an admin interface. Edit the `mysite/polls/admin.py` file and add the following to the bottom of the file:

```
from mysite.polls.models import Poll
from django.contrib import admin

admin.site.register(Poll)
```

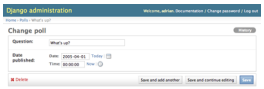
Now reload the Django admin page to see your changes. Note that you don't have to restart the development server -- the server will auto-reload your project, so any modifications code will be seen immediately in your browser.

# Explore the free admin functionality

Now that we've registered `Poll`, Django knows that it should be displayed on the admin index page:

Click "Polls." Now you're at the "change list" page for polls. This page displays all the polls in the database and lets you choose one to change it. There's the "What's up?" poll we created in the first tutorial:

Click the "What's up?" poll to edit it:



Things to note here:

- The form is automatically generated from the Poll model.
- The different model field types (`DateTimeField`, `CharField`) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each `DateTimeField` gets free JavaScript shortcuts. Dates get a "Today" shortcut and calendar popup, and times get a "Now" shortcut and a convenient popup that lists commonly entered times.

The bottom part of the page gives you a couple of options:

- Save -- Saves changes and returns to the change-list page for this type of object.
- Save and continue editing -- Saves changes and reloads the admin page for this object.
- Save and add another -- Saves changes and loads a new, blank form for this type of object.
- Delete -- Displays a delete confirmation page.

Change the "Date published" by clicking the "Today" and "Now" shortcuts. Then click "Save and continue editing." Then click "History" in the upper right. You'll see a page listing all changes made to this object via the Django admin, with the timestamp and username of the person who made the change:

# Customize the admin form

Take a few minutes to marvel at all the code you didn't have to write. When you call `admin.site.register(Poll)`, Django just lets you edit the object and "guess" at how to display it within the admin. Often you'll want to control how the admin looks and works. You'll do this by telling Django about the options you want when you register the object.

Let's see how this works by reordering the fields on the edit form. Replace the `admin.site.register(Poll)` line with:

```
class PollAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question']

admin.site.register(Poll, PollAdmin)
```

You'll follow this pattern -- create a model admin object, then pass it as the second argument to `admin.site.register()` -- any time you need to change the admin options for an object.

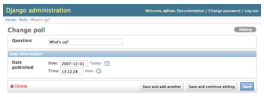
This particular change above makes the "Publication date" come before the "Question" field:


This isn't impressive with only two fields, but for admin forms with dozens of fields, choosing an intuitive order is an important usability detail.

And speaking of forms with dozens of fields, you might want to split the form up into fieldsets:

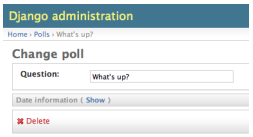
```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Poll, PollAdmin)
```

The first element of each tuple in `fieldsets` is the title of the fieldset. Here's what our form looks like now:



You can assign arbitrary HTML classes to each fieldset. Django provides a `"collapse"` class that displays a particular fieldset initially collapsed. This is useful when you have a long form that contains a number of fields that aren't commonly used:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
```




## Adding related objects

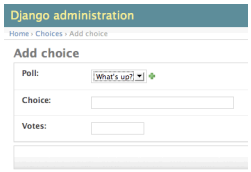
OK, we have our Poll admin page. But a `Poll` has multiple `Choices`, and the admin page doesn't display choices.

Yet.

There are two ways to solve this problem. The first register `Choice` with the admin just as we did with `Poll`. That's easy:

```
from mysite.polls.models import Choice

admin.site.register(Choice)
```

Now "Choices" is an available option in the Django admin. The "Add choice" form looks like this:




In that form, the "Poll" field is a select box containing every poll in the database. Django knows that a `ForeignKey` should be represented in the admin as a `<select>` box. In our case, only one poll exists at this point.

Also note the "Add Another" link next to "Poll." Every object with a `ForeignKey` relationship to another gets this for free. When you click "Add Another," you'll get a popup window with the "Add poll" form. If you add a poll in that window and click "Save," Django will save the poll to the database and dynamically add it as the selected choice on the "Add choice" form you're looking at.

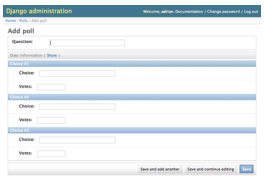
But, really, this is an inefficient way of adding Choice objects to the system. It'd be better if you could add a bunch of Choices directly when you create the Poll object. Let's make that happen.

Remove the `register()` call for the Choice model. Then, edit the `Poll` registration code to read:

```
class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

This tells Django: "Choice objects are edited on the Poll admin page. By default, provide enough fields for 3 choices."

Load the "Add poll" page to see how that looks:

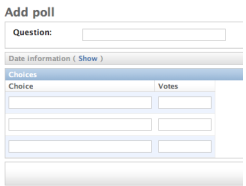



It works like this: There are three slots for related Choices -- as specified by `extra` -- and each time you come back to the "Change" page for an already-created object, you get another three extra slots.

One small problem, though. It takes a lot of screen space to display all the fields for entering related Choice objects. For that reason, Django offers a tabular way of displaying inline related objects; you just need to change the `ChoiceInline` declaration to read:

```
class ChoiceInline(admin.TabularInline):
    #...
```

With that `TabularInline` (instead of `StackedInline`), the related objects are displayed in a more compact, table-based format:




# Customize the admin change list

Now that the Poll admin page is looking good, let's make some tweaks to the "change list" page -- the one that displays all the polls in the system.

Here's what it looks like at this point:


By default, Django displays the `str()` of each object. But sometimes it'd be more helpful if we could display individual fields. To do that, use the `list_display` admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date')
```

Just for good measure, let's also include the `was_published_today` custom method from Tutorial 1:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date', 'was_published_today')
```

Now the poll change list page looks like this:

You can click on the column headers to sort by those values -- except in the case of the `was_published_today` header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for `was_published_today` is, by default, the name of the method (with underscores replaced with spaces). But you can change that by giving that method a `short_description` attribute:

```
def was_published_today(self):
    return self.pub_date.date() == datetime.date.today()
was_published_today.short_description = 'Published today?'
```

Let's add another improvement to the Poll change list page: Filters. Add the following line to `PollAdmin`:

```
list_filter = ['pub_date']
```

That adds a "Filter" sidebar that lets people filter the change list by the `pub_date` field:

The type of filter displayed depends on the type of field you're filtering on. Because `pub_date` is a DateTimeField, Django knows to give the default filter options for DateTimeFields: "Any date," "Today," "Past 7 days," "This month," "This year."

This is shaping up well. Let's add some search capability:

```
search_fields = ['question']
```

That adds a search box at the top of the change list. When somebody enters search terms, Django will search the `question` field. You can use as many fields as you'd like -- although because it uses a `LIKE` query behind the scenes, keep it reasonable, to keep your database happy.

Finally, because Poll objects have dates, it'd be convenient to be able to drill down by date. Add this line:

```
date_hierarchy = 'pub_date'
```

That adds hierarchical navigation, by date, to the top of the change list page. At top level, it displays all available years. Then it drills down to months and, ultimately, days.

Now's also a good time to note that change lists give you free pagination. The default is to display 50 items per page. Change-list pagination, search boxes, filters, date-hierarchies and column-header-ordering all work together like you think they should.

# Customize the admin look and feel

Clearly, having "Django administration" at the top of each admin page is ridiculous. It's just placeholder text.

That's easy to change, though, using Django's template system. The Django admin is powered by Django itself, and its interfaces use Django's own template system. (How meta!)

Open your settings file (`mysite/settings.py`, remember) and look at the `TEMPLATE_DIRS` setting. `TEMPLATE_DIRS` is a tuple of filesystem directories to check when loading Django templates. It's a search path.

By default, `TEMPLATE_DIRS` is empty. So, let's add a line to it, to tell Django where our templates live:

```
TEMPLATE_DIRS = (
    "/home/my_username/mytemplates", # Change this to your own directory.
)
```

Now copy the template `admin/base_site.html` from within the default Django admin template directory (`django/contrib/admin/templates`) into an `admin` subdirectory of whichever directory you're using in `TEMPLATE_DIRS`. For example, if your `TEMPLATE_DIRS` includes `"/home/my_username/mytemplates"`, as above, then copy `django/contrib/admin/templates/admin/base_site.html` to `/home/my_username/mytemplates/admin/base_site.html`. Don't forget that `admin` subdirectory.

Then, just edit the file and replace the generic Django text with your own site's name as you see fit.

Note that any of Django's default admin templates can be overridden. To override a template, just do the same thing you did with `base_site.html` -- copy it from the default directory into your custom directory, and make changes.

Astute readers will ask: But if `TEMPLATE_DIRS` was empty by default, how was Django finding the default admin templates? The answer is that, by default, Django automatically looks for a `templates/` subdirectory within each app package, for use as a fallback. See the *template loader documentation* for full information.

# Customize the admin index page

On a similar note, you might want to customize the look and feel of the Django admin index page.

By default, it displays all the apps in `INSTALLED_APPS` that have been registered with the admin application, in alphabetical order. You may want to make significant changes to the layout. After all, the index is probably the most important page of the admin, and it should be easy to use.

The template to customize is `admin/index.html`. (Do the same as with `admin/base_site.html` in the previous section -- copy it from the default directory to your custom template directory.) Edit the file, and you'll see it uses a template variable called `app_list`. That variable contains every installed Django app. Instead of using that, you can hard-code links to object-specific admin pages in whatever way you think is best.

When you're comfortable with the admin site, read *part 3 of this tutorial* to start working on public poll views.