Contents

[Django settings 4](#_Toc8140360)

[The basics 4](#_Toc8140361)

[Designating the settings 4](#_Toc8140362)

[**The import Search Path** 4](#_Toc8140363)

[The django-admin utility 4](#_Toc8140364)

[On the server (mod\_wsgi) 4](#_Toc8140365)

[Default settings 4](#_Toc8140366)

[Seeing which settings you’ve changed 4](#_Toc8140367)

[Using settings in Python code 4](#_Toc8140368)

[Altering settings at runtime 5](#_Toc8140369)

[Security 5](#_Toc8140370)

[Available settings 5](#_Toc8140371)

[Creating your own settings 5](#_Toc8140372)

[Using settings without setting DJANGO\_SETTINGS\_MODULE 5](#_Toc8140373)

[Custom default settings 5](#_Toc8140374)

[Required configure() or DJANGO\_SETTINGS\_MODULE 5](#_Toc8140375)

[django.setup() required for “standalone” Django usage 5](#_Toc8140376)

[Configure Django project for multiple environments? 6](#_Toc8140377)

[Django settings for multiple environments 6](#_Toc8140378)

[Starting a Django project 6](#_Toc8140379)

[Splitting requirements 6](#_Toc8140380)

[Base.txt 6](#_Toc8140381)

[Production.txt 6](#_Toc8140382)

[Test.txt 6](#_Toc8140383)

[Local.txt 6](#_Toc8140384)

[Splitting settings 7](#_Toc8140385)

[config.setting.base 7](#_Toc8140386)

[config.settings.local 7](#_Toc8140387)

[config.settings.production 7](#_Toc8140388)

[config.setting.test 7](#_Toc8140389)

[uwsgi.py and urls.py files 8](#_Toc8140390)

[Summary 8](#_Toc8140391)

[Django Group By 8](#_Toc8140392)

[your first Django app, part 1 10](#_Toc8140393)

[Creating a project 10](#_Toc8140394)

[The development server 11](#_Toc8140395)

[Changing the port 11](#_Toc8140396)

[Automatic reloading of runserver 11](#_Toc8140397)

[Creating the Polls app 11](#_Toc8140398)

[Projects vs. apps 11](#_Toc8140399)

[Write your first view 11](#_Toc8140400)

[When to use include() 12](#_Toc8140401)

[Page not found? 12](#_Toc8140402)

[path() argument: route 12](#_Toc8140403)

[path() argument: view 12](#_Toc8140404)

[path() argument: kwargs 12](#_Toc8140405)

[path() argument: name 12](#_Toc8140406)

[your first Django app, part 2 12](#_Toc8140407)

[Database setup 12](#_Toc8140408)

[For databases other than SQLite 12](#_Toc8140409)

[For the minimalists 13](#_Toc8140410)

[Creating models 13](#_Toc8140411)

[Philosophy 13](#_Toc8140412)

[Activating models 13](#_Toc8140413)

[Philosophy 13](#_Toc8140414)

[Playing with the API 14](#_Toc8140415)

[Introducing the Django Admin 15](#_Toc8140416)

[Philosophy 15](#_Toc8140417)

[Creating an admin user 15](#_Toc8140418)

[Start the development server 15](#_Toc8140419)

[Enter the admin site¶ 16](#_Toc8140420)

[Make the poll app modifiable in the admin 16](#_Toc8140421)

[Explore the free admin functionality 16](#_Toc8140422)

[your first Django app, part 3 16](#_Toc8140423)

[Overview 16](#_Toc8140424)

[Writing more views 17](#_Toc8140425)

[Write views that actually do something 17](#_Toc8140426)

[Template namespacing 17](#_Toc8140427)

[A shortcut: render() 18](#_Toc8140428)

[Raising a 404 error 18](#_Toc8140429)

[A shortcut: get\_object\_or\_404() 18](#_Toc8140430)

[Philosophy 18](#_Toc8140431)

[Use the template system 18](#_Toc8140432)

[Removing hardcoded URLs in templates 18](#_Toc8140433)

[Namespacing URL names 18](#_Toc8140434)

[your first Django app, part 4 19](#_Toc8140435)

[Write a simple form 19](#_Toc8140436)

[Use generic views: Less code is better 20](#_Toc8140437)

[Amend URLconf 20](#_Toc8140438)

[Amend views 20](#_Toc8140439)

[your first Django app, part 5 20](#_Toc8140440)

[Introducing automated testing 20](#_Toc8140441)

[What are automated tests? 20](#_Toc8140442)

[Why you need to create tests 20](#_Toc8140443)

[Tests will save you time 21](#_Toc8140444)

[Tests don’t just identify problems, they prevent them 21](#_Toc8140445)

[Tests make your code more attractive 21](#_Toc8140446)

[Tests help teams work together 21](#_Toc8140447)

[Basic testing strategies 21](#_Toc8140448)

[Writing our first test 21](#_Toc8140449)

[We identify a bug 21](#_Toc8140450)

[Create a test to expose the bug 21](#_Toc8140451)

[Running tests 21](#_Toc8140452)

[Fixing the bug 22](#_Toc8140453)

[More comprehensive tests 22](#_Toc8140454)

[Test a view 22](#_Toc8140455)

[A test for a view 22](#_Toc8140456)

[The Django test client 22](#_Toc8140457)

[Improving our view 22](#_Toc8140458)

[Testing our new view 23](#_Toc8140459)

[Testing the DetailView 23](#_Toc8140460)

[Ideas for more tests 24](#_Toc8140461)

[When testing, more is better 24](#_Toc8140462)

[Further testing 24](#_Toc8140463)

[What’s next? 24](#_Toc8140464)

[your first Django app, part 6 24](#_Toc8140465)

[Customize your *app’s* look and feel 24](#_Toc8140466)

[Static file namespacing 24](#_Toc8140467)

[Adding a background-image 25](#_Toc8140468)

[your first Django app, part 7 25](#_Toc8140469)

[Customize the admin form 25](#_Toc8140470)

[Adding related objects 25](#_Toc8140471)

[Customize the admin change list 26](#_Toc8140472)

[Customize the admin look and feel 27](#_Toc8140473)

[Customizing your *project’s* templates 27](#_Toc8140474)

[Organizing templates 27](#_Toc8140475)

[Where are the Django source files? 27](#_Toc8140476)

[Customizing your *application’s* templates 27](#_Toc8140477)

[Customize the admin index page 27](#_Toc8140478)

[What’s next? 28](#_Toc8140479)

[Advanced tutorial: How to write reusable apps 28](#_Toc8140480)

[Reusability matters 28](#_Toc8140481)

[Package? App? 28](#_Toc8140482)

[Your project and your reusable app 28](#_Toc8140483)

[Installing some prerequisites 28](#_Toc8140484)

[Packaging your app 28](#_Toc8140485)

[Using your own package 29](#_Toc8140486)

[Installing as a user library 29](#_Toc8140487)

[Publishing your app 29](#_Toc8140488)

[Installing Python packages with virtualenv 29](#_Toc8140489)

[What to read next 30](#_Toc8140490)

[Finding documentation 30](#_Toc8140491)

[How the documentation is organized 30](#_Toc8140492)

[How documentation is updated 30](#_Toc8140493)

[Where to get it 30](#_Toc8140494)

[Differences between versions 31](#_Toc8140495)

[Django 2.0 url() to path() cheatsheet 31](#_Toc8140496)

[DJANGO REST Framework 33](#_Toc8140497)

[Example 33](#_Toc8140498)

[Quickstart 33](#_Toc8140499)

[Project setup 33](#_Toc8140500)

[Serializers 33](#_Toc8140501)

[Views 34](#_Toc8140502)

[URLs 34](#_Toc8140503)

[Pagination 34](#_Toc8140504)

[Settings 34](#_Toc8140505)

[Testing our API 34](#_Toc8140506)

[API development using Django REST 35](#_Toc8140507)

[1. User Create View 35](#_Toc8140508)

[2. User Detail View 36](#_Toc8140509)

[3. User Update View 36](#_Toc8140510)

[4. User List View 36](#_Toc8140511)

[5. User DeleteView 36](#_Toc8140512)

[Build API with Django REST and Class-Based Views 36](#_Toc8140513)

[Setup 37](#_Toc8140514)

[settings.py 37](#_Toc8140515)

[Project Level URL Routing 38](#_Toc8140516)

[auth\_api/urls.py 38](#_Toc8140517)

[Making the Root View of the API 38](#_Toc8140518)

[auth\_api/views.py 38](#_Toc8140519)

[Configuring URLs for Users and Todos 38](#_Toc8140520)

[todos/urls.py 38](#_Toc8140521)

[users/urls.py 38](#_Toc8140522)

[Writing the Views 38](#_Toc8140523)

[todos/views.py: 39](#_Toc8140524)

[users/views.py 39](#_Toc8140525)

[Writing the Serializers 39](#_Toc8140526)

[Todos/serializers.py 39](#_Toc8140527)

[users/serializers.py 39](#_Toc8140528)

[Writing the Models 40](#_Toc8140529)

[todos/models.py 40](#_Toc8140530)

[users/models.py 40](#_Toc8140531)

[Bringing It All Together 40](#_Toc8140532)

[Conclusion 41](#_Toc8140533)

[Django REST: Beginners Guide 41](#_Toc8140534)

[Initial Setup 41](#_Toc8140535)

[Models 41](#_Toc8140536)

[Serialization 42](#_Toc8140537)

[URLs 43](#_Toc8140538)

[Browsable API 43](#_Toc8140539)

[Requests and Responses 43](#_Toc8140540)

[Adding information to our model 44](#_Toc8140541)

[Adding endpoints to our User models 45](#_Toc8140542)

[Associating Snippets with Users 45](#_Toc8140543)

[Updating our serializer 45](#_Toc8140544)

[Adding required permissions to views 45](#_Toc8140545)

[Adding login to the browsable API 46](#_Toc8140546)

[Object level permissions 46](#_Toc8140547)

[Root API Endpoint 47](#_Toc8140548)

[Highlighted Snippets Endpoint 47](#_Toc8140549)

[Hyperlinking our API 48](#_Toc8140550)

[Pagination 48](#_Toc8140551)

[Viewsets and Routers 49](#_Toc8140552)

[Schemas 49](#_Toc8140553)

[Command line client 49](#_Toc8140554)

[Authenticating our client 50](#_Toc8140555)

[Conclusion 50](#_Toc8140556)

[Design philosophies 51](#_Toc8140557)

[Overall 51](#_Toc8140558)

[Loose coupling 51](#_Toc8140559)

[Less code 51](#_Toc8140560)

[Quick development 51](#_Toc8140561)

[Don’t repeat yourself (DRY) 51](#_Toc8140562)

[Explicit is better than implicit 51](#_Toc8140563)

[Consistency 51](#_Toc8140564)

[Models 51](#_Toc8140565)

[Explicit is better than implicit 51](#_Toc8140566)

[Include all relevant domain logic 51](#_Toc8140567)

[Database API 51](#_Toc8140568)

[SQL efficiency 51](#_Toc8140569)

[Terse, powerful syntax 51](#_Toc8140570)

[Option to drop into raw SQL easily, when needed 51](#_Toc8140571)

[URL design 51](#_Toc8140572)

[Loose coupling 51](#_Toc8140573)

[Infinite flexibility 51](#_Toc8140574)

[Encourage best practices 51](#_Toc8140575)

[Definitive URLs 51](#_Toc8140576)

[Template system 51](#_Toc8140577)

[Separate logic from presentation 51](#_Toc8140578)

[Discourage redundancy 52](#_Toc8140579)

[Be decoupled from HTML 52](#_Toc8140580)

[XML should not be used for template languages 52](#_Toc8140581)

[Assume designer competence 52](#_Toc8140582)

[Treat whitespace obviously 52](#_Toc8140583)

[Don’t invent a programming language 52](#_Toc8140584)

[Safety and security 52](#_Toc8140585)

[Extensibility 52](#_Toc8140586)

[Views 52](#_Toc8140587)

[Simplicity 52](#_Toc8140588)

[Use request objects 52](#_Toc8140589)

[Loose coupling 52](#_Toc8140590)

[Differentiate between GET and POST 52](#_Toc8140591)

[Cache Framework 52](#_Toc8140592)

[Less code 52](#_Toc8140593)

[Consistency 52](#_Toc8140594)

[Extensibility 52](#_Toc8140595)

[Django Object-Relational Mapper 52](#_Toc8140596)

[Django ORM resources 52](#_Toc8140597)

[Django migrations resources 53](#_Toc8140598)

[Shortcomings in Django ORM: look at Peewee 53](#_Toc8140599)

[The Django ORM inconsistent API 53](#_Toc8140600)

[Lack of composability 54](#_Toc8140601)

[Learning from Django 54](#_Toc8140602)

[Peewee is more consistent *and* expressive 54](#_Toc8140603)

["Fixing" Django 55](#_Toc8140604)

[What do you think? 55](#_Toc8140605)

[Links 55](#_Toc8140606)

[Documenting your API 55](#_Toc8140607)

[Built-in API documentation 55](#_Toc8140608)

[Installation 55](#_Toc8140609)

[Documenting your views 55](#_Toc8140610)

[documentation API Reference 56](#_Toc8140611)

[Customising code samples 57](#_Toc8140612)

[Third party packages 57](#_Toc8140613)

[Self describing APIs 58](#_Toc8140614)

[The hypermedia approach 59](#_Toc8140615)

[Django vs SQLAlchemy – Which Python ORM is better? 59](#_Toc8140616)

[What’s an ORM? 59](#_Toc8140617)

[Comparing Django vs SQLAlchemy 59](#_Toc8140618)

[Active Record vs Data Mapper 59](#_Toc8140619)

[Working with Complex Queries 59](#_Toc8140620)

[Primary Key Automatic Generation 59](#_Toc8140621)

[Autocommit 59](#_Toc8140622)

[Supported Databases 59](#_Toc8140623)

[Learning Curve 59](#_Toc8140624)

[Community Size 59](#_Toc8140625)

[Performance 60](#_Toc8140626)

[Summary 60](#_Toc8140627)

[SQLAlchemy and You 60](#_Toc8140628)

[Design Differences 60](#_Toc8140629)

[SQLAlchemy's Complexity 60](#_Toc8140630)

[The Session — The Heart of the ORM 60](#_Toc8140631)

[The Declarative Extension 61](#_Toc8140632)

[Basic Models 61](#_Toc8140633)

[Many-to-One Relationships 61](#_Toc8140634)

[Many-To-Many 62](#_Toc8140635)

[Translating Queries From Django To SQLAlchemy 62](#_Toc8140636)

[Inserting Entries 62](#_Toc8140637)

[Deleting Entries 62](#_Toc8140638)

[Updating Entries 62](#_Toc8140639)

[Primary Key Queries 62](#_Toc8140640)

[General Query Syntax 62](#_Toc8140641)

[Date Based Queries 63](#_Toc8140642)

[Sorting 63](#_Toc8140643)

[Aggregates 63](#_Toc8140644)

[Joins 63](#_Toc8140645)

[Why Consider SQLAlchemy? 64](#_Toc8140646)

[Web Server Gateway Interface 64](#_Toc8140647)

[The Many Frameworks Problem 64](#_Toc8140648)

[The HTTP Protocol 64](#_Toc8140649)

[Hello World! 65](#_Toc8140650)

[The environ Dictionary 66](#_Toc8140651)

[Summary 67](#_Toc8140652)

[What is Middleware? 67](#_Toc8140653)

[Getting Started 67](#_Toc8140654)

[Middleware Chains 68](#_Toc8140655)

[Error Handling 68](#_Toc8140656)

[Configuration 70](#_Toc8140657)

[The Many Frameworks Problem Revisited 70](#_Toc8140658)

# Django settings

A Django settings file contains all the configuration of your Django installation. This document explains how settings work and which settings are available.

## The basics

A settings file is just a Python module with module-level variables.

Here are a couple of example settings:

ALLOWED\_HOSTS = ['www.example.com']

DEBUG = False

DEFAULT\_FROM\_EMAIL = 'webmaster@example.com'

**Note**: If you set [DEBUG](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-DEBUG) to False, you also need to properly set the [ALLOWED\_HOSTS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-ALLOWED_HOSTS) setting. Because a settings file is a Python module, the following apply:

* It doesn’t allow for Python syntax errors.
* It can assign settings dynamically using normal Python syntax. For example:
* MY\_SETTING = [str(i) for i in range(30)]
* It can import values from other settings files.

## Designating the settings

You have to tell Django which settings you’re using. Do this by using an environment variable, DJANGO\_SETTINGS\_MODULE. The value of DJANGO\_SETTINGS\_MODULE should be in Python path syntax, e.g. mysite.settings. Note that the settings module should be on the Python [import search path](https://www.diveinto.org/python3/your-first-python-program.html#importsearchpath).

**The import Search Path**

Before this goes any further, I want to briefly mention the library search path. Python looks in several places when you try to import a module. Specifically, it looks in all the directories defined in sys.path. This is just a list, and you can easily view it or modify it with standard list methods. (You’ll learn more about lists in [Native Datatypes](https://www.diveinto.org/python3/native-datatypes.html#lists).)

>>> import sys ①

>>> sys.path ②

['',   
 '/usr/lib/python31.zip',   
 '/usr/lib/python3.1',  
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',   
 '/usr/lib/python3.1/lib-dynload',   
 '/usr/lib/python3.1/dist-packages',   
 '/usr/local/lib/python3.1/dist-packages']

>>> sys ③

<module 'sys' (built-in)>

>>> sys.path.insert(0, '/home/mark/diveintopython3/examples') ④

>>> sys.path ⑤

['/home/mark/diveintopython3/examples',   
 '',   
 '/usr/lib/python31.zip',   
 '/usr/lib/python3.1',   
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',   
 '/usr/lib/python3.1/lib-dynload',   
 '/usr/lib/python3.1/dist-packages',   
 '/usr/local/lib/python3.1/dist-packages']

|  |  |
| --- | --- |
| ① | Importing the sys module makes all of its functions and attributes available. |
| ② | sys.path is a list of directory names that constitute the current search path. (Yours will look different, depending on your operating system, what version of Python you’re running, and where it was originally installed.) Python will look through these directories (in this order) for a .py file whose name matches what you’re trying to import. |
| ③ | Actually, the truth is more complicated than that, because not all modules are stored as .py files. Some are *built-in modules*; they are actually baked right into Python itself. Built-in modules behave just like regular modules, but their Python source code is not available, because they are not written in Python! (Like Python itself, these built-in modules are written in C.) |
| ④ | You can add a new directory to Python’s search path at runtime by adding the directory name to sys.path, and then Python will look in that directory as well, whenever you try to import a module. The effect lasts as long as Python is running. |
| ⑤ | By using sys.path.insert(0, *new\_path*), you inserted a new directory as the first item of the sys.path list, and therefore at the beginning of Python’s search path. This is almost always what you want. In case of naming conflicts (for example, if Python ships with version 2 of a particular library but you want to use version 3), this ensures that your modules will be found and used instead of the modules that came with Python. |

## The django-admin utility

With [django-admin](https://docs.djangoproject.com/en/2.2/ref/django-admin/), you can either set the environment variable or explicitly pass in the settings module each time you run the utility.

**Example (Unix Bash shell)**:

export DJANGO\_SETTINGS\_MODULE=mysite.settings

django-admin runserver

**Example (Windows shell)**:

set DJANGO\_SETTINGS\_MODULE=mysite.settings

django-admin runserver

Use the --settings command-line argument to specify manually:

django-admin runserver --settings=mysite.settings

## On the server (mod\_wsgi)

In your live server environment, you’ll need to tell your WSGI application what settings file to use. Do that with os.environ:

import os

os.environ['DJANGO\_SETTINGS\_MODULE'] = 'mysite.settings'

Read the [Django mod\_wsgi documentation](https://docs.djangoproject.com/en/2.2/howto/deployment/wsgi/modwsgi/) for more information and other common elements to a Django WSGI application.

## Default settings

A Django settings file doesn’t have to define any settings if it doesn’t need to. Each setting has a sensible default value. These defaults live in the module django/conf/global\_settings.py.

Here’s the algorithm Django uses in compiling settings:

* Load settings from global\_settings.py.
* Load settings from the specified settings file, overriding the global settings as necessary.

Note that a settings file should *not* import from global\_settings, because that’s redundant.

## Seeing which settings you’ve changed

There’s an easy way to view which of your settings deviate from the default settings. The command python manage.py diffsettings displays differences between the current settings file and Django’s default settings. For more, see the [diffsettings](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-diffsettings) documentation.

## Using settings in Python code

In your Django apps, use settings by importing the object django.conf.settings. Example:

from django.conf import settings

if settings.DEBUG:

# Do something

Note that django.conf.settings isn’t a module – it’s an object. So importing individual settings is not possible:

from django.conf.settings import DEBUG # This won't work.

Also note that your code should *not* import from either global\_settings or your own settings file. django.conf.settings abstracts the concepts of default settings and site-specific settings; it presents a single interface. It also decouples the code that uses settings from the location of your settings.

## Altering settings at runtime

You shouldn’t alter settings in your applications at runtime. For example, don’t do this in a view:

from django.conf import settings

settings.DEBUG = True # Don't do this!

The only place you should assign to settings is in a settings file.

## Security

Because a settings file contains sensitive information, such as the database password, you should make every attempt to limit access to it. For example, change its file permissions so that only you and your Web server’s user can read it. This is especially important in a shared-hosting environment.

## Available settings

For a full list of available settings, see the [settings reference](https://docs.djangoproject.com/en/2.2/ref/settings/).

## Creating your own settings

There’s nothing stopping you from creating your own settings, for your own Django apps. Just follow these guidelines:

* Setting names must be all uppercase.
* Don’t reinvent an already-existing setting.

For settings that are sequences, Django itself uses lists, but this is only a convention.

## Using settings without setting DJANGO\_SETTINGS\_MODULE

You might want to bypass the DJANGO\_SETTINGS\_MODULE environment variable. For example, if you’re using the template system by itself, you likely don’t want to have to set up an environment variable pointing to a settings module. You can configure Django’s settings manually. Do this by calling:

django.conf.settings.configure(*default\_settings*, *\*\*settings*)

Example:

from django.conf import settings

settings.configure(DEBUG=True)

Pass configure() as many keyword arguments as you’d like, with each keyword argument representing a setting and its value. Each argument name should be all uppercase, with the same name as the settings described above. If a particular setting is not passed to configure() and is needed at some later point, Django will use the default setting value. Configuring Django in this fashion is mostly necessary – and, indeed, recommended – when you’re using a piece of the framework inside a larger application. Consequently, when configured via settings.configure(), Django will not make any modifications to the process environment variables (see the documentation of [TIME\_ZONE](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TIME_ZONE) for why this would normally occur).

## Custom default settings

If you’d like default values to come from somewhere other than django.conf.global\_settings, you can pass in a module or class that provides the default settings as the default\_settings argument (or as the first positional argument) in the call to configure(). In this example, default settings are taken from myapp\_defaults, and the [DEBUG](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-DEBUG) setting is set to True, regardless of its value in myapp\_defaults:

from django.conf import settings

from myapp import myapp\_defaults

settings.configure(default\_settings=myapp\_defaults, DEBUG=True)

The following example, which uses myapp\_defaults as a positional argument, is equivalent:

settings.configure(myapp\_defaults, DEBUG=True)

Normally, you will not need to override the defaults in this fashion. The Django defaults are sufficiently tame that you can safely use them. Be aware that if you do pass in a new default module, it entirely *replaces* the Django defaults, so you must specify a value for every possible setting that might be used in that code you are importing. Check in django.conf.settings.global\_settings for the full list.

## Required configure() or DJANGO\_SETTINGS\_MODULE

If you’re not setting the DJANGO\_SETTINGS\_MODULE environment variable, you *must* call configure() at some point before using any code that reads settings. If you don’t set DJANGO\_SETTINGS\_MODULE and don’t call configure(), Django will raise an ImportError exception the first time a setting is accessed. If you set DJANGO\_SETTINGS\_MODULE, access settings values somehow, *then* call configure(), Django will raise a RuntimeError indicating that settings have already been configured. There is a property just for this purpose: For example:

from django.conf import settings

if not settings.configured:

settings.configure(myapp\_defaults, DEBUG=True)

Also, it’s an error to call configure() more than once, or to call configure() after any setting has been accessed. It boils down to this: Use one of either configure() or DJANGO\_SETTINGS\_MODULE. Not both, and not neither.

## django.setup() required for “standalone” Django usage

If you’re using components of Django “standalone” – for example, writing a Python script which loads some Django templates and renders them, or uses the ORM to fetch some data – there’s one more step you’ll need in addition to configuring settings. After you’ve either set [DJANGO\_SETTINGS\_MODULE](https://docs.djangoproject.com/en/2.2/topics/settings/#envvar-DJANGO_SETTINGS_MODULE) or called configure(), you’ll need to call [django.setup()](https://docs.djangoproject.com/en/2.2/ref/applications/#django.setup) to load your settings and populate Django’s application registry. For example:

import django

from django.conf import settings

from myapp import myapp\_defaults

settings.configure(default\_settings=myapp\_defaults, DEBUG=True)

django.setup()

# Now this script or any imported module can use any part of Django it needs.

from myapp import models

Note that calling django.setup() is only necessary if your code is truly standalone. When invoked by your Web server, or through [django-admin](https://docs.djangoproject.com/en/2.2/ref/django-admin/), Django will handle this for you. django.setup() may only be called once. Therefore, avoid putting reusable application logic in standalone scripts so that you have to import from the script elsewhere in your application. If you can’t avoid that, put the call to :

django.setup() inside an if block:

if \_\_name\_\_ == '\_\_main\_\_':

import django

django.setup()

See also: [The Settings Reference](https://docs.djangoproject.com/en/2.2/ref/settings/) Contains the complete list of core and contrib app settings.

# Configure Django project for multiple environments?

[Django](https://blog.apptension.com/tag/django) [Link](https://blog.apptension.com/author/krystian-hanek/)

If you’re about to configure Django settings for multiple environments, you need to think ahead. Your simple project may grow significantly and you’ll have to introduce changes in order to run your app on different environments. In this article, I will show you how to configure [Django](https://blog.apptension.com/2017/09/13/rest-api-using-django-rest-framework/) settings for multiple environments, based on [The Twelve-Factor App](https://12factor.net) methodology for building software-as-a-service apps. I have also used [Cookiecutter Django framework](https://github.com/pydanny/cookiecutter-django) by Pydanny. This project uses several third-party tools including PostgreSQL, Sentry, AWS, WhiteNoise, Gunicorn, Redis, Anymail. **TL;DR** If you’d like to just take a quick glance at the code, take a look at this SlideShare prezentation. Below, I’ll explain how to configure Django project step by step. But before we set up the new project, let’s tackle this question:

## Django settings for multiple environments

The no 1 reason to configure Django settings for multiple environments is that when you first start a new project, it lacks such arrangement. Not having the bundles split makes it difficult to configure the project later without having to alter the code. Also, there are no dedicated solutions for production like dedicated path for admin panel, logging errors (e.g., Sentry), cache configuration (memcache/redis), saving the data uploaded to cloud by the user (S3), HSTS or secure cookies. Testing environment, on the other hand, lacks dedicated solutions either, including turning off debugging templates, in-memory caching, sending mails to console, password hasher, storing the templates. As a result, setting Django project for multiple environments gives you:

* more manageable code with fewer duplications
* more accurate settings depending on environment type

## Starting a Django project

First, you need to set up our new Django project. To do it, install virtualenv/virtualenvwrapper and Django: pip install Django==1.11.5 or whatever Django version you want to use for your project. Then, create a new project: django-admin: django-admin startproject djangohotspot.

At this point, your project should look like this:

(djangohotspot) ╭ ~/Workspace/

╰$ tree djangohotspot

djangohotspot

├── djangohotspot

│   ├── \_\_init\_\_.py

│   ├── settings.py

│   ├── urls.py

│   └── wsgi.py

└── manage.py

Now you need to set it up for multiple environments. You do it by splitting the requirements first.

## Splitting requirements

Create base.txt that will cover the common requirements. Then write down the environment-specific requirements in separate files:

* local.txt
* production.txt
* test.txt

The structure of the project is now as follows:

(djangohotspot) ╭ ~/Workspace/

╰$ tree djangohotspot

djangohotspot

├── djangohotspot

│   ├── \_\_init\_\_.py

│   ├── settings.py

│   ├── urls.py

│   └── wsgi.py

├── manage.py

└── requirements

    ├── base.txt

    ├── local.txt

    ├── production.txt

    └── test.txt

### Base.txt

django==1.11.5

# Configuration

django-environ==0.4.4

whitenoise==3.3.0

# Models

django-model-utils==3.0.0

# Images

Pillow==4.2.1

# Password storage

argon2-cffi==16.3.0

# Python-PostgreSQL Database Adapter

psycopg2==2.7.3.1

# Unicode slugification

awesome-slugify==1.6.5

# Time zones support

pytz==2017.2

# Redis support

django-redis==4.8.0

redis>=2.10.5

### Production.txt

-r base.txt

# WSGI Handler

gevent==1.2.2

gunicorn==19.7.1

# Static and Media Storage

boto3==1.4.7

django-storages==1.6.5

# Email backends for Mailgun, Postmark,

# SendGrid and more

django-anymail==0.11.1

# Raven is the Sentry client

raven==6.1.0

### Test.txt

-r base.txt

coverage==4.4.1

flake8==3.4.1

factory-boy==2.9.2

# pytest

pytest-cov==2.4.0

pytest-django==3.1.2

pytest-factoryboy==1.3.1

pytest-mock==1.6.0

pytest-sugar==0.9.0

### Local.txt

This file combines production and test files:

-r test.txt

-r production.txt

django-extensions==1.9.0

ipdb==0.10.3

It’s time to configure the settings of each environment.

## Splitting settings

First, you need to remove settings.py from the main folder of your Django app : djangohotspot/djangohotspot/settings.py

and create new Python module named config, where you create another module, settings, where all the settings files will be stored. The structure of your project has changed and should look like this now:

(djangohotspot) ╭ ~/Workspace/

╰$ tree djangohotspot

djangohotspot

├── config

│   ├── \_\_init\_\_.py

│   └── settings

│       ├── base.py

│       ├── \_\_init\_\_.py

│       ├── local.py

│       ├── production.py

│       └── test.py

├── djangohotspot

│   ├── \_\_init\_\_.py

│   ├── urls.py

│   └── wsgi.py

├── manage.py

└── requirements

    ├── base.txt

    ├── local.txt

    ├── production.txt

    └── test.txt

### config.setting.base

To configure settings in base.py in this example, I have used the [django-environ](https://github.com/joke2k/django-environ) library.

# django-environ:3 folders back=> ‘djangohotspot/’

ROOT\_DIR = environ.Path(\_\_file\_\_) - 3

# path for django apps

APPS\_DIR = ROOT\_DIR.path('djangohotspot')

INSTALLED\_APPS = DJANGO\_APPS + THIRD\_PARTY\_APPS + LOCAL\_APPS

# To define function for unicode-supported Slug

AUTOSLUG\_SLUGIFY\_FUNCTION = 'slugify.slugify'

DATABASES = { 'default': env.db('DATABASE\_URL', default='[postgres:///djangohotspot](postgres://djangohotspot)'), }

# allows you to open and commit transaction when

# there are no exceptions. This could affect the

# performance negatively for traffic-heavy apps

DATABASES['default']['ATOMIC\_REQUESTS'] = True

EMAIL\_BACKEND = env('DJANGO\_EMAIL\_BACKEND', default='django.core.mail.backends.smtp.EmailBackend')

ADMIN\_URL = env('DJANGO\_ADMIN\_URL', default=r'^admin/')

PASSWORD\_HASHERS = ['django.contrib.auth.hashers.Argon2PasswordHasher', (...)] # add this object at the beginning of the list

### config.settings.local

Configuring local settings, you need to import base settings:

from .base import \*

Now, add debug toolbar:

MIDDLEWARE += ['debug\_toolbar.middleware.DebugToolbarMiddleware', ]

INSTALLED\_APPS += ['debug\_toolbar', ]

DEBUG\_TOOLBAR\_CONFIG = {

  'DISABLE\_PANELS': [ 'debug\_toolbar.panels. redirects.RedirectsPanel', ],

  'SHOW\_TEMPLATE\_CONTEXT': True,

}

Define allowed IP addresses:

INTERNAL\_IPS = ['127.0.0.1']

And add Django extension:

INSTALLED\_APPS += ['django\_extensions', ]

### config.settings.production

In production settings, focus on the security for your project.

# security configuration

SECURE\_HSTS\_SECONDS = 60

SECURE\_HSTS\_INCLUDE\_SUBDOMAINS = env.bool( 'DJANGO\_SECURE\_HSTS\_INCLUDE\_SUBDOMAINS', default=True)

SECURE\_CONTENT\_TYPE\_NOSNIFF = env.bool( 'DJANGO\_SECURE\_CONTENT\_TYPE\_NOSNIFF', default=True)

SECURE\_BROWSER\_XSS\_FILTER = True

SESSION\_COOKIE\_SECURE = True

SESSION\_COOKIE\_HTTPONLY = True

SECURE\_SSL\_REDIRECT = env.bool('DJANGO\_SECURE\_SSL\_REDIRECT', default=True)

CSRF\_COOKIE\_SECURE = True

CSRF\_COOKIE\_HTTPONLY = True

X\_FRAME\_OPTIONS = 'DENY'

ADMIN\_URL = env('DJANGO\_ADMIN\_URL')

At this point it’s worth to add DJANGO\_ADMIN\_URL to the production settings. Change it from default to avoid attack attempts on the default URL admin panel. Next, add your domains:

ALLOWED\_HOSTS = env.list('DJANGO\_ALLOWED\_HOSTS', default=['djangohotspot.pl', ])

And add a Gunicorn:

INSTALLED\_APPS += ['gunicorn', ]

Finally, add django-storage for AWS:

INSTALLED\_APPS += ['storages', ]

AWS\_ACCESS\_KEY\_ID = env('DJANGO\_AWS\_ACCESS\_KEY\_ID')

AWS\_SECRET\_ACCESS\_KEY = env('DJANGO\_AWS\_SECRET\_ACCESS\_KEY')

AWS\_STORAGE\_BUCKET\_NAME = env('DJANGO\_AWS\_STORAGE\_BUCKET\_NAME')

AWS\_AUTO\_CREATE\_BUCKET = True

AWS\_QUERYSTRING\_AUTH = False

AWS\_EXPIRY = 60 \* 60 \* 24 \* 7

MEDIA\_URL = '[https://s3.amazonaws.com/%s/](https://s3.amazonaws.com/%25s/)' % AWS\_STORAGE\_BUCKET\_NAME

DEFAULT\_FILE\_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'

To efficiently track errors you may add Sentry:

INSTALLED\_APPS += ['raven.contrib.django.raven\_compat', ]

RAVEN\_MIDDLEWARE = ['raven.contrib.django.raven\_compat.middleware.SentryResponseErrorIdMiddleware']

MIDDLEWARE = RAVEN\_MIDDLEWARE + MIDDLEWARE

SENTRY\_DSN = env('DJANGO\_SENTRY\_DSN')

SENTRY\_CLIENT = env('DJANGO\_SENTRY\_CLIENT', default='raven.contrib.django.raven\_compat.DjangoClient')

SENTRY\_CELERY\_LOGLEVEL = env.int('DJANGO\_SENTRY\_LOG\_LEVEL', logging.INFO)

RAVEN\_CONFIG = {

  'CELERY\_LOGLEVEL': env.int('DJANGO\_SENTRY\_LOG\_LEVEL', logging.INFO),

  'DSN': SENTRY\_DSN,

}

And if you want to serve static files, add WhiteNoise:

WHITENOISE\_MIDDLEWARE = ['whitenoise.middleware.WhiteNoiseMiddleware', ]

MIDDLEWARE = WHITENOISE\_MIDDLEWARE + MIDDLEWARE

STATICFILES\_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

### config.setting.test

Configuring test settings, start with turning debugging off:

DEBUG = False

TEMPLATES[0]['OPTIONS']['debug'] = False

Store sent mails in memory available in django.core.mail.outbox:

EMAIL\_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'

Set the cache:

CACHES = {

  'default': {

     'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',

     'LOCATION': ''

  }

}

Set the password hasher to speed up the tests:

PASSWORD\_HASHERS = ['django.contrib.auth.hashers.MD5PasswordHasher', ]

If you use Django templates, set them to be stored in memory:

TEMPLATES[0]['OPTIONS']['loaders'] = [

  ['django.template.loaders.cached.Loader', [

     'django.template.loaders.filesystem.Loader',

     'django.template.loaders.app\_directories.Loader', ]

  , ]

, ]

## uwsgi.py and urls.py files

Because we split the main settings file into dedicated files with configuration for each environment, we need to point a file which is used by default when not clearly indicated. In urls.py we define the 4xx and 5xx pages. Also add debug toolbar here. Move uwsgi.py and urls.py from djangohotspot/djangohotspot catalogue to config module and add following changes to config.settings:

WSGI\_APPLICATION = 'config.wsgi.application'

ROOT\_URLCONF = 'config.urls'

At the end of the config.urls file add the following code to debug 4xx and 5xx pages:

if settings.DEBUG:

     urlpatterns += [

          url(r'^400/$',

              default\_views.bad\_request,

              kwargs={'exception': Exception('Bad Request!')}),

          url(r'^403/$',

              default\_views.permission\_denied,

              kwargs={'exception': Exception('Permission Denied')}),

          url(r'^404/$',

              default\_views.page\_not\_found,

              kwargs={'exception': Exception('Page not Found')}),

          url(r'^500/$',

              default\_views.server\_error),

     ]

if 'debug\_toolbar' in settings.INSTALLED\_APPS:

     import debug\_toolbar

          urlpatterns = [

          url(r'^\_\_debug\_\_/', include(debug\_toolbar.urls)),

] + urlpatterns

In our example, **config.uwsgi** file will look like this:

import os

import sys

from django.core.wsgi import get\_wsgi\_application

app\_path = os.path.dirname(os.path.abspath (\_\_file\_\_)).replace('/config', '')

sys.path.append(os.path.join(app\_path, 'djangohotspot'))

if os.environ.get('DJANGO\_SETTINGS\_MODULE') == 'config.settings.production':

  from raven.contrib.django.raven\_compat. middleware.wsgi import Sentry

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "config.settings.production")

application = get\_wsgi\_application()

if os.environ.get('DJANGO\_SETTINGS\_MODULE') == 'config.settings.production':

  application = Sentry(application)

## Summary

As you’ve seen in this article, setting a Django project for multiple environments is a toilsome task. But trust me, it pays off quickly once you start working on the project. From this point on, you can think of some containerization with Docker, which will give you portability and easiness of setup for your project regardless the environment it will be run on.

# Django Group By

This tutorial is about how to implement SQL-like group by queries using the Django ORM. It’s a fairly common operation, specially for those who are familiar with SQL. The Django ORM is actually an abstraction layer, that let us play with the database as it was object-oriented but in the end it’s just a relational database and all the operations are translated into SQL statements. Most of the work can be done retrieving the raw data from the database, and playing with it in the Python side, grouping the data in dictionaries, iterating through it, making sums, averages and what not. But the database is a very powerful tool and do much more than simply storing the data, and often you can do the work much faster directly in the database. Generally speaking, when you start doing group by queries, you are no longer interested in each model instances (or in a table row) details, but you want extract new information from your dataset, based on some common aspects shared between the model instances. Let’s have a look in an example:

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

name = models.CharField(max\_length=30)

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

name = models.CharField(max\_length=30)

country = models.ForeignKey(Country)

population = models.PositiveIntegerField()

And the raw data stored in the database:

|  |  |  |  |
| --- | --- | --- | --- |
| cities | | | |
| id | **name** | **country\_id** | **population** |
| 1 | Tokyo | 28 | 36,923,000 |
| 2 | Shanghai | 13 | 34,000,000 |
| 3 | Jakarta | 19 | 30,000,000 |
| 4 | Seoul | 21 | 25,514,000 |
| 5 | Guangzhou | 13 | 25,000,000 |
| 6 | Beijing | 13 | 24,900,000 |
| 7 | Karachi | 22 | 24,300,000 |
| 8 | Shenzhen | 13 | 23,300,000 |
| 9 | Delhi | 25 | 21,753,486 |
| 10 | Mexico City | 24 | 21,339,781 |
| 11 | Lagos | 9 | 21,000,000 |
| 12 | São Paulo | 1 | 20,935,204 |
| 13 | Mumbai | 25 | 20,748,395 |
| 14 | New York City | 20 | 20,092,883 |
| 15 | Osaka | 28 | 19,342,000 |
| 16 | Wuhan | 13 | 19,000,000 |
| 17 | Chengdu | 13 | 18,100,000 |
| 18 | Dhaka | 4 | 17,151,925 |
| 19 | Chongqing | 13 | 17,000,000 |
| 20 | Tianjin | 13 | 15,400,000 |
| 21 | Kolkata | 25 | 14,617,882 |
| 22 | Tehran | 11 | 14,595,904 |
| 23 | Istanbul | 2 | 14,377,018 |
| 24 | London | 26 | 14,031,830 |
| 25 | Hangzhou | 13 | 13,400,000 |
| 26 | Los Angeles | 20 | 13,262,220 |
| 27 | Buenos Aires | 8 | 13,074,000 |
| 28 | Xi'an | 13 | 12,900,000 |
| 29 | Paris | 6 | 12,405,426 |
| 30 | Changzhou | 13 | 12,400,000 |
| 31 | Shantou | 13 | 12,000,000 |
| 32 | Rio de Janeiro | 1 | 11,973,505 |
| 33 | Manila | 18 | 11,855,975 |
| 34 | Nanjing | 13 | 11,700,000 |
| 35 | Rhine-Ruhr | 16 | 11,470,000 |
| 36 | Jinan | 13 | 11,000,000 |
| 37 | Bangalore | 25 | 10,576,167 |
| 38 | Harbin | 13 | 10,500,000 |
| 39 | Lima | 7 | 9,886,647 |
| 40 | Zhengzhou | 13 | 9,700,000 |
| 41 | Qingdao | 13 | 9,600,000 |
| 42 | Chicago | 20 | 9,554,598 |
| 43 | Nagoya | 28 | 9,107,000 |
| 44 | Chennai | 25 | 8,917,749 |
| 45 | Bangkok | 15 | 8,305,218 |
| 46 | Bogotá | 27 | 7,878,783 |
| 47 | Hyderabad | 25 | 7,749,334 |
| 48 | Shenyang | 13 | 7,700,000 |
| 49 | Wenzhou | 13 | 7,600,000 |
| 50 | Nanchang | 13 | 7,400,000 |
| 51 | Hong Kong | 13 | 7,298,600 |
| 52 | Taipei | 29 | 7,045,488 |
| 53 | Dallas–Fort Worth | 20 | 6,954,330 |
| 54 | Santiago | 14 | 6,683,852 |
| 55 | Luanda | 23 | 6,542,944 |
| 56 | Houston | 20 | 6,490,180 |
| 57 | Madrid | 17 | 6,378,297 |
| 58 | Ahmedabad | 25 | 6,352,254 |
| 59 | Toronto | 5 | 6,055,724 |
| 60 | Philadelphia | 20 | 6,051,170 |
| 61 | Washington, D.C. | 20 | 6,033,737 |
| 62 | Miami | 20 | 5,929,819 |
| 63 | Belo Horizonte | 1 | 5,767,414 |
| 64 | Atlanta | 20 | 5,614,323 |
| 65 | Singapore | 12 | 5,535,000 |
| 66 | Barcelona | 17 | 5,445,616 |
| 67 | Munich | 16 | 5,203,738 |
| 68 | Stuttgart | 16 | 5,200,000 |
| 69 | Ankara | 2 | 5,150,072 |
| 70 | Hamburg | 16 | 5,100,000 |
| 71 | Pune | 25 | 5,049,968 |
| 72 | Berlin | 16 | 5,005,216 |
| 73 | Guadalajara | 24 | 4,796,050 |
| 74 | Boston | 20 | 4,732,161 |
| 75 | Sydney | 10 | 5,000,500 |
| 76 | San Francisco | 20 | 4,594,060 |
| 77 | Surat | 25 | 4,585,367 |
| 78 | Phoenix | 20 | 4,489,109 |
| 79 | Monterrey | 24 | 4,477,614 |
| 80 | Inland Empire | 20 | 4,441,890 |
| 81 | Rome | 3 | 4,321,244 |
| 82 | Detroit | 20 | 4,296,611 |
| 83 | Milan | 3 | 4,267,946 |
| 84 | Melbourne | 10 | 4,650,000 |
| countries | |
| id | **name** |
| 1 | Brazil |
| 2 | Turkey |
| 3 | Italy |
| 4 | Bangladesh |
| 5 | Canada |
| 6 | France |
| 7 | Peru |
| 8 | Argentina |
| 9 | Nigeria |
| 10 | Australia |
| 11 | Iran |
| 12 | Singapore |
| 13 | China |
| 14 | Chile |
| 15 | Thailand |
| 16 | Germany |
| 17 | Spain |
| 18 | Philippines |
| 19 | Indonesia |
| 20 | United States |
| 21 | South Korea |
| 22 | Pakistan |
| 23 | Angola |
| 24 | Mexico |
| 25 | India |
| 26 | United Kingdom |
| 27 | Colombia |
| 28 | Japan |
| 29 | Taiwan |

Considering the whole dataset, to know the total of habitants in all the 84 cities, we could perhaps use an aggregate query:

**from** **django.db.models** **import** Sum

City.objects.aggregate(Sum('population'))

{'population\_\_sum': 970880224} *# 970,880,224*

Or the average population in the top 84 cities:

**from** **django.db.models** **import** Avg

City.objects.aggregate(Avg('population'))

{'population\_\_avg': 11558097.904761905} *# 11,558,097.90*

What if we now wanted to see the total population, but aggregated by the country instead? Not the whole dataset. In this case we no longer can use aggregate, instead we will be using annotate.

The aggregate clause is terminal, it returns a Python dictionary, meaning you can’t append any queryset methods. Also, it will always return a single result. So to get the population sum by country, using aggregate:

Don't

**from** **django.db.models** **import** Sum

**for** country **in** Country.objects.all():

result = City.objects.filter(country=country) .aggregate(Sum('population'))

**print** '{}: {}'.format(country.name, result['population\_\_sum'])

*# Output:*

*# -------*

*# Brazil: 38676123*

*# Turkey: 19527090*

*# Italy: 8589190*

*# Bangladesh: 17151925*

*# Canada: 6055724*

*# France: 12405426*

*# Peru: 9886647*

*# Argentina: 13074000*

*# Nigeria: 21000000*

*# Australia: 9650500*

*# Iran: 14595904*

*# ...*

While the result is correct, we needed to execute **30** different queries in the database. And we’ve lost some of the capabilities of the ORM, such as ordering this result set. Perhaps the data would be more interesting if we could order by the country with the most population for example. A better way to do it is using annotate, which will be translated as a **group by** query in the database:

Do

City.objects.values('country\_\_name') .annotate(Sum('population'))

[

{'country\_\_name': u'Angola', 'population\_\_sum': 6542944},

{'country\_\_name': u'Argentina', 'population\_\_sum': 13074000},

{'country\_\_name': u'Australia', 'population\_\_sum': 9650500},

{'country\_\_name': u'Bangladesh', 'population\_\_sum': 17151925},

{'country\_\_name': u'Brazil', 'population\_\_sum': 38676123},

'...(remaining elements truncated)...'

]

Much better, right? Now if we wanted to order by the country population, we can use an alias to make it look cleaner and to use in the order\_by() clause:

City.objects.values('country\_\_name') \

.annotate(country\_population=Sum('population')) \

.order\_by('-country\_population')

[

{'country\_\_name': u'China', 'country\_population': 309898600},

{'country\_\_name': u'United States', 'country\_population': 102537091},

{'country\_\_name': u'India', 'country\_population': 100350602},

{'country\_\_name': u'Japan', 'country\_population': 65372000},

{'country\_\_name': u'Brazil', 'country\_population': 38676123},

'...(remaining elements truncated)...'

]

Here is how the last SQL query looks like:

SELECT "core\_country"."name", SUM("core\_city"."population") AS "country\_population"

FROM "core\_city" INNER JOIN "core\_country" ON ("core\_city"."country\_id" = "core\_country"."id")

GROUP BY "core\_country"."name"

ORDER BY "country\_population" DESC

Now an important thing to note here: it only makes sense adding in the values() clause, the data that can be grouped. Every field you add to the values() clause, will be used to create the group by query. Look at this queryset:

City.objects.values('name', 'country\_\_name').annotate(Sum('population'))

The resulting SQL query would be:

SELECT "core\_city"."name", "core\_country"."name", SUM("core\_city"."population") AS "population\_\_sum"

FROM "core\_city" INNER JOIN "core\_country" ON ("core\_city"."country\_id" = "core\_country"."id")

GROUP BY "core\_city"."name", "core\_country"."name"

This would have no effect, because all the city names are unique, and they can’t be grouped (the database will try to group it, but each “group” will have only 1 row/instance). We can see it simply by performing a count on each queryset:

City.objects.values('name', 'country\_\_name').annotate(Sum('population')).count()

84

City.objects.values('country\_\_name').annotate(Sum('population')).count()

29

That’s what I meant when I said in the beginning of the post that, you are no longer interested in the details of each row. When we group by country to get the sum of the population, we lost the details of the cities (at least in the query result). Sometimes it makes sense to have more than one value in the values() clause. For example if our database was composed by City / State / Country. Then we could group by using .values('state\_\_name', 'country\_\_name'). This way you would have the population by country. And you would avoid States from different countries (with the same name) to be grouped together. The values you generate on the database, using the annotate clause, can also be used to filter data. Usually in the database we use the HAVING function, which makes it very idiomatic. You can read the query like it was plain English. Now, in the Django side, it’s a simple filter. For example, let’s say we want to see the total population by country, but only those countries where the total population is greater than 50,000,000:

City.objects.values('country\_\_name') \

.annotate(country\_population=Sum('population')) \

.filter(country\_population\_\_gt=50000000) \

.order\_by('-country\_population')

[

{'country\_\_name': u'China', 'country\_population': 309898600},

{'country\_\_name': u'United States', 'country\_population': 102537091},

{'country\_\_name': u'India', 'country\_population': 100350602},

{'country\_\_name': u'Japan', 'country\_population': 65372000}

]

And finally the SQL query:

SELECT "core\_country"."name", SUM("core\_city"."population") AS "country\_population"

FROM "core\_city" INNER JOIN "core\_country" ON ("core\_city"."country\_id" = "core\_country"."id")

GROUP BY "core\_country"."name" HAVING SUM("core\_city"."population") > 50000000

ORDER BY "country\_population" DESC

I hope you found this small tutorial helpful! If you have any questions, please leave a comment below!

# your first Django app, part 1

[Let’s learn by example](https://www.djangoproject.com/start/).

Throughout this tutorial, we’ll walk you through the creation of a basic poll application. It’ll consist of two parts:

* A public site that lets people view polls and vote in them.
* An admin site that lets you add, change, and delete polls.

We’ll assume you have [Django installed](https://docs.djangoproject.com/en/2.2/intro/install/) already. You can tell Django is installed and which version by running the following command in a shell prompt (indicated by the $ prefix):

$ python -m django --version

If Django is installed, you should see the version of your installation. If it isn’t, you’ll get an error telling “No module named django”. This tutorial is written for **Django 2.2**, which supports Python 3.5 and later. If the Django version doesn’t match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you’re using an older version of Python, check [What Python version can I use with Django?](https://docs.djangoproject.com/en/2.2/faq/install/#faq-python-version-support) to find a compatible version of Django. See [How to install Django](https://docs.djangoproject.com/en/2.2/topics/install/) for advice on how to remove older versions of Django and install a newer one. **Where to get help**: If you’re having trouble going through this tutorial, please post a message to [django-users](https://docs.djangoproject.com/en/2.2/internals/mailing-lists/#django-users-mailing-list) or drop by [#django on irc.freenode.net](irc://irc.freenode.net/django) to chat with other Django users who might be able to help.

## Creating a project

If this is your first time using Django, you’ll have to take care of some initial setup. Namely, you’ll need to auto-generate some code that establishes a Django [project](https://docs.djangoproject.com/en/2.2/glossary/#term-project) – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, cd into a directory where you’d like to store your code, then run the following command:

$ django-admin startproject mysite

This will create a mysite directory in your current directory. If it didn’t work, see [Problems running django-admin](https://docs.djangoproject.com/en/2.2/faq/troubleshooting/#troubleshooting-django-admin).

**Note**: You’ll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like django (which will conflict with Django itself) or test (which conflicts with a built-in Python package).

Where should this code live? If your background is in plain old PHP (with no use of modern frameworks), you’re probably used to putting code under the Web server’s document root (in a place such as /var/www). With Django, you don’t do that. It’s not a good idea to put any of this Python code within your Web server’s document root, because it risks the possibility that people may be able to view your code over the Web. That’s not good for security. Put your code in some directory **outside** of the document root, such as /home/mycode. Let’s look at what [startproject](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-startproject) created:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

These files are:

* The outer mysite/ root directory is just a container for your project. Its name doesn’t matter to Django; you can rename it to anything you like.
* manage.py: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about manage.py in [django-admin and manage.py](https://docs.djangoproject.com/en/2.2/ref/django-admin/).
* The inner mysite/ directory is the actual Python package for your project. Its name is the Python package name you’ll need to use to import anything inside it (e.g. mysite.urls).
* mysite/\_\_init\_\_.py: An empty file that tells Python that this directory should be considered a Python package. Rread [more about packages](https://docs.python.org/3/tutorial/modules.html#tut-packages) in the official Python docs.
* mysite/settings.py: Settings/configuration for this Django project. [Django settings](https://docs.djangoproject.com/en/2.2/topics/settings/) will tell you how settings work.
* mysite/urls.py: The URL declarations for this Django project; a “table of contents” of your Django-powered site. You can read more about URLs in [URL dispatcher](https://docs.djangoproject.com/en/2.2/topics/http/urls/).
* mysite/wsgi.py: An entry-point for WSGI[[1]](#footnote-1)-compatible web servers to serve your project. See [How to deploy with WSGI](https://docs.djangoproject.com/en/2.2/howto/deployment/wsgi/) for more details.

## The development server

Let’s verify your Django project works. Change into the outer mysite directory, and run the following commands:

$ python manage.py runserver

You’ll see the following output on the command line:

Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.

Run 'python manage.py migrate' to apply them.

April 18, 2019 - 15:50:53

Django version 2.2, using settings 'mysite.settings'

Starting development server at <http://127.0.0.1:8000/>

Quit the server with CONTROL-C.

**Note**: Ignore the warning about unapplied database migrations for now; we’ll deal with the database shortly. You’ve started the *Django development server*, a lightweight Web server written purely in Python. We’ve included this with Django so you can develop things rapidly, without having to deal with configuring a production server – such as Apache – until you’re ready for production. Now’s a good time to note: **don’t** use this server in anything resembling a production environment. It’s intended only for use while developing. (We’re in the business of making Web frameworks, not Web servers.) Now that the server’s running, visit <http://127.0.0.1:8000/> with your Web browser.See a “Congratulations!” page, with a rocket taking off. It worked!

### Changing the port

By default, the [runserver](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-runserver) command starts the development server on the internal IP at port 8000. If you want to change the server’s port, pass it as a command-line argument. For instance, this command starts the server on port 8080:

$ python manage.py runserver 8080

If you want to change the server’s IP, pass it along with the port. For example, to listen on all available public IPs (which is useful if you are running Vagrant or want to show off your work on other computers on the network), use:

$ python manage.py runserver 0:8000

**0** is a shortcut for **0.0.0.0**. Full docs for the development server can be found in the [runserver](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-runserver) reference.

### Automatic reloading of [runserver](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-runserver)

The development server automatically reloads Python code for each request as needed. You don’t need to restart the server for code changes to take effect. However, some actions like adding files don’t trigger a restart, so you’ll have to restart the server in these cases.

## Creating the Polls app

Now that your environment – a “project” – is set up, you’re set to start doing work. Each application you write in Django consists of a Python package that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.

### Projects vs. apps

What’s the difference between a project and an app? An **app** is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app. A **project** is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.Your apps can live anywhere on your [Python path](https://docs.python.org/3/tutorial/modules.html#tut-searchpath). In this tutorial, our poll app is next to your manage.py file so that it can be imported as its own top-level module, rather than a submodule of mysite. To create your app, make sure you’re in the same directory as manage.py and type this command:

$ python manage.py startapp polls

That’ll create a directory polls, which is laid out like this:

polls/

\_\_init\_\_.py

admin.py

apps.py

migrations/

\_\_init\_\_.py

models.py

tests.py

views.py

This directory structure will house the poll application.

## Write your first view

Let’s write the first view. Open the file polls/views.py and put the following Python code in it:

polls/views.py

from django.http import HttpResponse

def index(request):

return HttpResponse("Hello, world. You're at the polls index.")

This is the simplest view possible in Django. To call the view, we need to map it to a URL - and for this we need a URLconf. To create a URLconf in the polls directory, create a file called urls.py. Your app directory should now look like:

polls/

\_\_init\_\_.py

admin.py

apps.py

migrations/

\_\_init\_\_.py

models.py

tests.py

urls.py

views.py

In the polls/urls.py file include the following code:

polls/urls.py

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

]

The next step is to point the root URLconf at the polls.urls module. In mysite/urls.py, add an import for django.urls.include and insert an [include()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.include) in the urlpatterns list, so you have:

mysite/urls.py

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path('polls/', include('polls.urls')),

path('admin/', admin.site.urls),

]

The [include()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.include) function allows referencing other URLconfs. Whenever Django encounters [include()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.include), it chops off whatever part of the URL matched up to that point and sends remaining string to the included URLconf for further processing. The idea behind [include()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.include) is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (polls/urls.py), they can be placed under “/polls/”, or under “/fun\_polls/”, or under “/content/polls/”, or other path root, and the app will still work.

### When to use [include()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.include)

You should always use include() when you include other URL patterns. admin.site.urls is the only exception to this.

You have now wired an index view into the URLconf. Verify it’s working with the following command:

$ python manage.py runserver

Go to <http://localhost:8000/polls/> in your browser, and you should see the text “*Hello, world. You’re at the polls index.*”, which you defined in the index view.

### Page not found?

If you get an error page here, check that you’re going to <http://localhost:8000/polls/> and not <http://localhost:8000/>.

The [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) function is passed four arguments, two required: route and view, and two optional: kwargs, and name. At this point, it’s worth reviewing what these arguments are for.

### [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) argument: route

route is a string that contains a URL pattern. When processing a request, Django starts at the first pattern in urlpatterns and makes its way down the list, comparing the requested URL against each pattern until it finds one that matches. Patterns don’t search GET and POST parameters, or the domain name. F. ex., in request to https://www.example.com/myapp/, URLconf will look for myapp/. In: https://www.example.com/myapp/?page=3, the URLconf will also look for myapp/.

### [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) argument: view

When Django finds a matching pattern, it calls the specified view function with an [HttpRequest](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest) object as the first argument and any “captured” values from the route as keyword arguments. We’ll give an example of this in a bit.

### [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) argument: kwargs

Arbitrary keyword arguments can be passed in a dictionary to the target view.

### [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) argument: name

Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This allows you to make global changes to the URL patterns of your project while only touching a single file. When you’re comfortable with the basic request and response flow, read [part 2 of this tutorial](https://docs.djangoproject.com/en/2.2/intro/tutorial02/) to start working with the database.

# your first Django app, part 2

This tutorial begins where [Tutorial 1](https://docs.djangoproject.com/en/2.2/intro/tutorial01/) left off. We’ll setup the database, create your first model, and get a quick introduction to Django’s automatically-generated admin site.

## Database setup

Now, open up mysite/settings.py. It’s a normal Python module with module-level variables representing Django settings. By default, the configuration uses **SQLit**e. If you’re new to databases, or you’re just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won’t need to install anything else to support your database. For real projects, use a more scalable database like **PostgreSQL**, to avoid database-switching headaches down the road. If you wish to use another database, install the appropriate [database bindings](https://docs.djangoproject.com/en/2.2/topics/install/#database-installation) and change the following keys in the [DATABASES](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-DATABASES) 'default' item to match your database connection settings:

* [ENGINE](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-DATABASE-ENGINE) –'django.db.backends.sqlite3', 'django.db.backends.postgresql', 'django.db.backends.mysql', or 'django.db.backends.oracle'. Other backends are [also available](https://docs.djangoproject.com/en/2.2/ref/databases/#third-party-notes).
* [NAME](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-NAME) – The name of your database. If you’re using SQLite, the database will be a file on your computer; in that case, [NAME](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-NAME) should be the full absolute path, including filename, of that file. The default value, os.path.join(BASE\_DIR, 'db.sqlite3'), will store the file in your project directory.

If you are not using SQLite, additional settings such as [USER](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-USER), [PASSWORD](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-PASSWORD), and [HOST](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-HOST) must be added. For more details, see the reference documentation for [DATABASES](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-DATABASES).

### For databases other than SQLite

If you’re using a database besides SQLite, make sure you’ve created a database by this point with “CREATE DATABASE database\_name;” within your database’s interactive prompt. Ensure that the database user provided in mysite/settings.py has “create database” privileges. This allows automatic creation of a [test database](https://docs.djangoproject.com/en/2.2/topics/testing/overview/#the-test-database) which will be needed in a later tutorial.

If you’re using SQLite, you don’t need to create anything beforehand - the database file will be created automatically when it is needed. While you’re editing mysite/settings.py, set [TIME\_ZONE](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TIME_ZONE) to your time zone. Also, note the [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS) setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects. By default, [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS) contains the following apps from Django:

* [django.contrib.admin](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#module-django.contrib.admin): AQdmin site.
* [django.contrib.auth](https://docs.djangoproject.com/en/2.2/topics/auth/#module-django.contrib.auth): Authentication system.
* [django.contrib.contenttypes](https://docs.djangoproject.com/en/2.2/ref/contrib/contenttypes/#module-django.contrib.contenttypes): Framework for content types.
* [django.contrib.sessions](https://docs.djangoproject.com/en/2.2/topics/http/sessions/#module-django.contrib.sessions): Session framework.
* [django.contrib.messages](https://docs.djangoproject.com/en/2.2/ref/contrib/messages/#module-django.contrib.messages): Messaging framework.
* [django.contrib.staticfiles](https://docs.djangoproject.com/en/2.2/ref/contrib/staticfiles/#module-django.contrib.staticfiles): Framework to manage static files.

These applications are included by default as a convenience for the common case. Some applications use min. 1 database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

$ python manage.py migrate

The [migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate) command looks at the [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS) setting and creates any necessary database tables according to the database settings in your mysite/settings.py file and the database migrations shipped with the app (we’ll cover those later). You’ll see a message for each migration it applies. If you’re interested, run the command-line client for your database and type \dt (PostgreSQL), SHOW TABLES; (MySQL), .schema (SQLite), or SELECT TABLE\_NAME FROM USER\_TABLES; (Oracle) to display the tables Django created.

### For the minimalists

Like we said above, the default applications are included for the common case, but not everybody needs them. If you don’t need any or all of them, feel free to comment-out or delete the appropriate line(s) from [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS) before running [migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate). The [migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate) command will only run migrations for apps in [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS).

## Creating models

Now we’ll define your models – essentially, your database layout, with additional metadata.

### Philosophy

A model is the single, definitive source of truth about your data. It contains the essential fields and behaviors of the data you’re storing. Django follows the [DRY Principle](https://docs.djangoproject.com/en/2.2/misc/design-philosophies/#dry). The goal is to define your data model in one place and automatically derive things from it. This includes the migrations - unlike in Ruby On Rails, for example, migrations are entirely derived from your models file, and are essentially just a history that Django can roll through to update your database schema to match your current models. In our simple poll app, we’ll create two models: A **Question** has a question and a publication date. A **Choice** has 2 fields: the text of the choice and a vote tally. Each Choice is associated with a Question. These concepts are represented by simple Python classes. Edit the polls/models.py file as:

polls/models.py

from django.db import models

class Question(models.Model):

question\_text = models.CharField(max\_length=200)

pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

question = models.ForeignKey(Question, on\_delete=models.CASCADE)

choice\_text = models.CharField(max\_length=200)

votes = models.IntegerField(default=0)

The code is straightforward. Each model is represented by a class that subclasses [django.db.models.Model](https://docs.djangoproject.com/en/2.2/ref/models/instances/#django.db.models.Model). Each model has a number of class variables, each of which represents a database field in the model. Each field is represented by an instance of a [Field](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.Field) class – e.g., [CharField](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.CharField) for character fields; [DateTimeField](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.DateTimeField) for datetimes. This tells Django what type of data each field holds. The name of each [Field](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.Field) instance (e.g. question\_text or pub\_date) is the field’s name, in machine-friendly format. You’ll use this value in your Python code, and your database will use it as the column name. You can use an optional first positional argument to a [Field](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.Field) to designate a human-readable name. That’s used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn’t provided, Django will use the machine-readable name. In this example, we’ve only defined a human-readable name for Question.pub\_date. For all other fields in this model, the field’s machine-readable name will suffice as its human-readable name. Some [Field](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.Field) classes have required arguments. [CharField](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.CharField), for example, requires that you give it a [max\_length](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.CharField.max_length). That’s used not only in the database schema, but in validation, as we’ll soon see. A [Field](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.Field) can also have various optional arguments; in this case, we’ve set the [default](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.Field.default) value of votes to 0. Finally, note a relationship is defined, using [ForeignKey](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.ForeignKey). That tells Django each Choice is related to a single Question. Django supports all the common database relationships: many-to-one, many-to-many, and one-to-one.

## Activating models

That small bit of model code gives Django a lot of information. With it, Django is able to:

* Create a **database schem**a (CREATE TABLE statements)
* Create a **Python database-access API** for accessing Question and Choice objects.

But first we need to tell our project that the polls app is installed.

### Philosophy

Django apps are “pluggable”: You can use an app in multiple projects, and you can distribute apps, because they don’t have to be tied to a given Django installation. To include the app in our project, we need to add a reference to its configuration class in the [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS) setting. The PollsConfig class is in the polls/apps.py file; its dotted path: 'polls.apps.PollsConfig'. Edit the mysite/settings.py file and add that dotted path to the [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS) setting. It’ll look like this:

mysite/settings.py

INSTALLED\_APPS = [

'polls.apps.PollsConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Now Django will include the polls app. Run new command:

$ python manage.py makemigrations polls

You should see something similar to the following:

Migrations for 'polls':

polls/migrations/0001\_initial.py:

- Create model Choice

- Create model Question

- Add field question to choice

By running makemigrations, you’re telling Django that you’ve made some changes to your models (you’ve made new ones) and that you’d like the changes to be stored as a *migration*.

**Migrations** are how Django stores changes to your models (and thus your database schema) - they’re just files on disk. You can read the migration for your new model if you like; it’s the file polls/migrations/0001\_initial.py. Don’t worry, you’re not expected to read them every time Django makes one, but they’re designed to be human-editable in case you want to manually tweak how Django changes things. There’s a command that will run the migrations for you and manage your database schema automatically - that’s called [migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate). But first, let’s see what SQL that migration would run. The [sqlmigrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-sqlmigrate) command takes migration names and returns their SQL:

$ python manage.py sqlmigrate polls 0001

You should see something similar to the following:

BEGIN;

--

-- Create model Choice

--

CREATE TABLE "polls\_choice" (

"id" serial NOT NULL PRIMARY KEY,

"choice\_text" varchar(200) NOT NULL,

"votes" integer NOT NULL

);

--

-- Create model Question

--

CREATE TABLE "polls\_question" (

"id" serial NOT NULL PRIMARY KEY,

"question\_text" varchar(200) NOT NULL,

"pub\_date" timestamp with time zone NOT NULL

);

--

-- Add field question to choice

--

ALTER TABLE "polls\_choice" ADD COLUMN "question\_id" integer NOT NULL;

ALTER TABLE "polls\_choice" ALTER COLUMN "question\_id" DROP DEFAULT;

CREATE INDEX "polls\_choice\_7aa0f6ee" ON "polls\_choice" ("question\_id");

ALTER TABLE "polls\_choice"

ADD CONSTRAINT "polls\_choice\_question\_id\_246c99a640fbbd72\_fk\_polls\_question\_id"

FOREIGN KEY ("question\_id")

REFERENCES "polls\_question" ("id")

DEFERRABLE INITIALLY DEFERRED;

COMMIT;

Note the following:

* The exact output will vary depending on the database you are using. The example above is generated for PostgreSQL.
* Table names are automatically generated by combining the name of the app (polls) and the lowercase name of the model – question and choice. (You can override this behavior.)
* Primary keys (IDs) are added automatically. (You can override this, too.)
* By convention, Django appends "\_id" to the foreign key field name. (Yes, you can override this, as well.)
* The foreign key relationship is made explicit by a FOREIGN KEY constraint. Don’t worry about the DEFERRABLE parts; (to tell PostgreSQL to not enforce the foreign key until the end of the transaction).
* It’s tailored to the database you’re using, so database-specific field types such as auto\_increment (MySQL), serial (PostgreSQL), or integer primary key autoincrement (SQLite) are handled automatically. Same goes for the quoting of field names – e.g., using double quotes or single quotes.
* The [sqlmigrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-sqlmigrate) command doesn’t actually run the migration on your database - it just prints it to the screen so that you can see what SQL Django thinks is required. It’s useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

If you’re interested, you can also run [python manage.py check](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-check); this checks for any problems in your project without making migrations or touching the database. Now, run [migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate) again to create those model tables in your database:

$ python manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, contenttypes, polls, sessions

Running migrations:

Rendering model states... DONE

Applying polls.0001\_initial... OK

The [migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate) command takes all the migrations that haven’t been applied (Django tracks which ones are applied using a special table in your database called django\_migrations) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database. Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data. We’ll cover them in more depth in a later part of the tutorial, but for now, remember the three-step guide to making model changes:

* Change your models (in models.py).
* Run [python manage.py makemigrations](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-makemigrations) to create migrations for those changes
* Run [python manage.py migrate](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate) to apply changes to database

The reason that there are separate commands to make and apply migrations is because you’ll commit migrations to your version control system and ship them with your app; they not only make your development easier, they’re also usable by other developers and in production. Read the [django-admin documentation](https://docs.djangoproject.com/en/2.2/ref/django-admin/) for full information on what the manage.py utility can do.

## Playing with the API

Now, let’s hop into the interactive Python shell and play with the free API Django. To invoke the Python shell, use this command:

$ python manage.py shell

We’re using this instead of simply typing “python”, because manage.py sets the DJANGO\_SETTINGS\_MODULE environment variable, which gives Django the Python import path to your mysite/settings.py file. Once you’re in the shell, explore the [database API](https://docs.djangoproject.com/en/2.2/topics/db/queries/):

>>> from polls.models import Choice, Question # Import the model classes we just wrote.

# No questions are in the system yet.

>>> Question.objects.all()

<QuerySet []>

# Create a new Question.

# Support for time zones is enabled in the default settings file, so

# Django expects a datetime with tzinfo for pub\_date. Use timezone.now()

# instead of datetime.datetime.now() and it will do the right thing.

>>> from django.utils import timezone

>>> q = Question(question\_text="What's new?", pub\_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.

>>> q.save()

# Now it has an ID.

>>> q.id

1

# Access model field values via Python attributes.

>>> q.question\_text

"What's new?"

>>> q.pub\_date

datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().

>>> q.question\_text = "What's up?"

>>> q.save()

# objects.all() displays all the questions in the database.

>>> Question.objects.all()

<QuerySet [<Question: Question object (1)>]>

Wait a minute. <Question: Question object (1)> isn’t a helpful representation of this object. Let’s fix that by editing the Question model (in the polls/models.py file) and adding a [\_\_str\_\_()](https://docs.djangoproject.com/en/2.2/ref/models/instances/#django.db.models.Model.__str__) method to both Question and Choice:

polls/models.py

from django.db import models

class Question(models.Model):

# ...

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

return self.question\_text

class Choice(models.Model):

# ...

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

return self.choice\_text

It’s important to add [\_\_str\_\_()](https://docs.djangoproject.com/en/2.2/ref/models/instances/#django.db.models.Model.__str__) methods to your models, because objects’ representations are used throughout Django’s auto-generated admin. Note these are normal Python methods. Let’s add a custom method, just for demonstration:

polls/models.py

import datetime

from django.db import models

from django.utils import timezone

class Question(models.Model):

# ...

def was\_published\_recently(self):

return self.pub\_date >= timezone.now() - datetime.timedelta(days=1)

Note the addition of import datetime and from django.utils import timezone, to reference Python’s standard [datetime](https://docs.python.org/3/library/datetime.html#module-datetime) module and Django’s time-zone-related utilities in [django.utils.timezone](https://docs.djangoproject.com/en/2.2/ref/utils/#module-django.utils.timezone), respectively. If you aren’t familiar with time zone handling in Python, you can learn more in the [time zone support docs](https://docs.djangoproject.com/en/2.2/topics/i18n/timezones/). Save these changes and start a new Python interactive shell by running python manage.py shell again:

>>> from polls.models import Choice, Question

# Make sure our \_\_str\_\_() addition worked.

>>> Question.objects.all()

<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by keyword arguments.

>>> Question.objects.filter(id=1)

<QuerySet [<Question: What's up?>]>

>>> Question.objects.filter(question\_text\_\_startswith='What')

<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.

>>> from django.utils import timezone

>>> current\_year = timezone.now().year

>>> Question.objects.get(pub\_date\_\_year=current\_year)

<Question: What's up?>

# Requesting non-existing ID will raise exception.

>>> Question.objects.get(id=2)

Traceback (most recent call last):

...

DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a shortcut for primary-key exact lookups.

# The following is identical to Question.objects.get(id=1).

>>> Question.objects.get(pk=1)

<Question: What's up?>

# Make sure our custom method worked.

>>> q = Question.objects.get(pk=1)

>>> q.was\_published\_recently()

True

# Give the Question a couple of Choices. The ‘create call’ constructs a new choice object, does the INSERT statement, adds the choice to the set of available choices and returns the new Choice object. Django creates

a set to hold the "other side" of a ForeignKey relation

(e.g. a question's choice)- accessible via the API.

>>> q = Question.objects.get(pk=1)

# Display choices from related object set -- none so far.

>>> q.choice\_set.all()

<QuerySet []>

# Create three choices.

>>> q.choice\_set.create(choice\_text='Not much', votes=0)

<Choice: Not much>

>>> q.choice\_set.create(choice\_text='The sky', votes=0)

<Choice: The sky>

>>> c = q.choice\_set.create(choice\_text='Just hacking again', votes=0)

# Choice objects have API access to Question objects.

>>> c.question

<Question: What's up?>

# Vice versa: Question objects access to Choice objects.

>>> q.choice\_set.all()

<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

>>> q.choice\_set.count()

3

# API automatically follows relationships as far as you need. Use double underscores to separate relationships.

This works as many levels deep as you want; there's no limit. Find all Choices for any question whose pub\_date is in this year (reusing the 'current\_year' variable we created above).

>>>Choice.objects.filter(question\_\_pub\_date\_\_year=current\_year)

<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

# Let's delete one of the choices. Use delete() for that.

>>> c = q.choice\_set.filter(choice\_text\_\_startswith='Just hacking')

>>> c.delete()

For more information on model relations, see [Accessing related objects](https://docs.djangoproject.com/en/2.2/ref/models/relations/). For more on how to use double underscores to perform field lookups via the API, see [Field lookups](https://docs.djangoproject.com/en/2.2/topics/db/queries/#field-lookups-intro). For full details on the database API, see our [Database API reference](https://docs.djangoproject.com/en/2.2/topics/db/queries/).

## Introducing the Django Admin

### 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 intended to be used by site visitors. It’s for site managers.

### Creating an admin user

First we’ll need to create a user who can login to the admin site. Run the following command:

$ python manage.py createsuperuser

Enter your desired username and press enter.

Username: admin

You will then be prompted for your desired email address:

Email address: admin@example.com

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

Password: \*\*\*\*\*\*\*\*\*\*

Password (again): \*\*\*\*\*\*\*\*\*

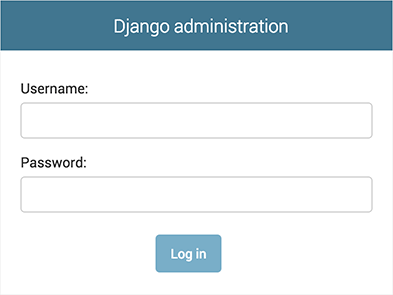
Superuser created successfully.

### Start the development server

The Django admin site is activated by default. Start development server and explore it. If the server is not running start it:

$ python manage.py runserver

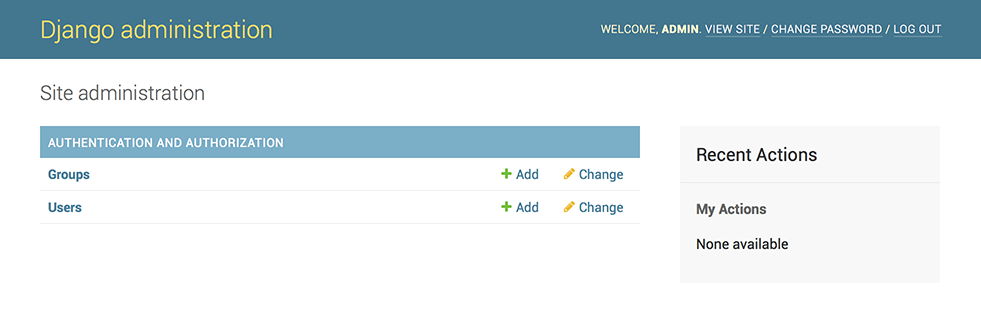
Now, open a Web browser and go to “/admin/” on your local domain – e.g., <http://127.0.0.1:8000/admin/>. You should see the admin’s login screen:



Since [translation](https://docs.djangoproject.com/en/2.2/topics/i18n/translation/) is turned on by default, the login screen may be displayed in your own language, depending on your browser’s settings and if Django has a translation for this language.

### Enter the admin site[¶](https://docs.djangoproject.com/en/2.2/intro/tutorial02/#enter-the-admin-site)

Now, try logging in with the superuser account you created in the previous step. You should see the Django admin index page:



You should see a few types of editable content: groups and users. They are provided by [django.contrib.auth](https://docs.djangoproject.com/en/2.2/topics/auth/#module-django.contrib.auth), the authentication framework shipped by Django.

### 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 Question objects have an admin interface. To do this, open the polls/admin.py file, and edit it to look like this:

polls/admin.py

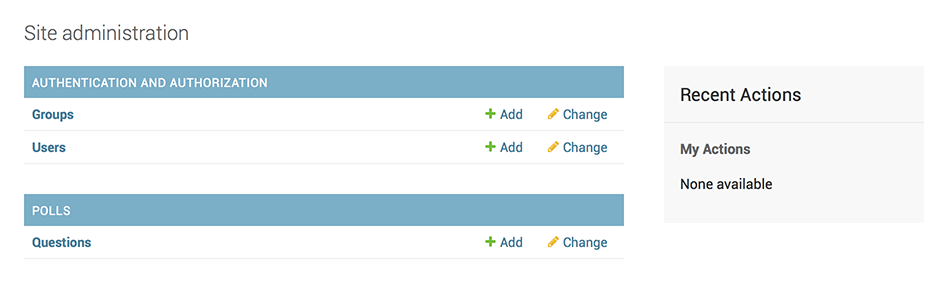
from django.contrib import admin

from .models import Question

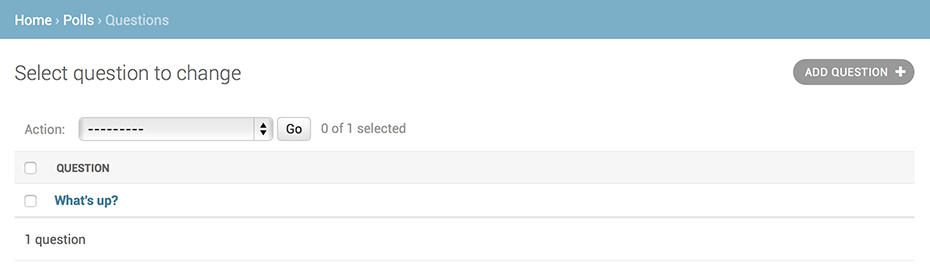
admin.site.register(Question)

### Explore the free admin functionality

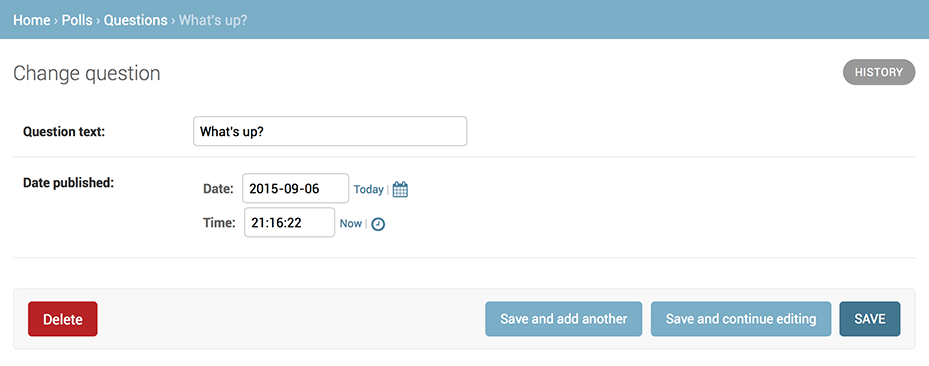
Now that we’ve registered Question, Django knows that it should be displayed on the admin index page:



Click “Questions”. Now you’re at the “change list” page for questions. This page displays all the questions in the database and lets you choose one to change it. There’s the “What’s up?” question we created earlier:



Click the “What’s up?” question to edit it:



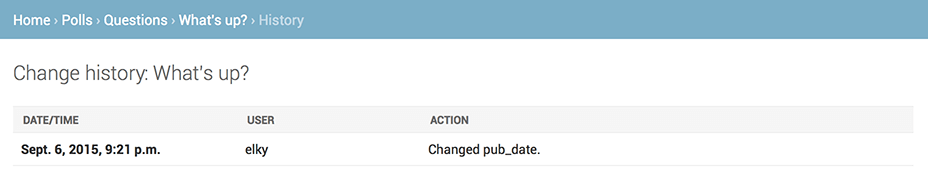
Things to note here:

* The form is automatically generated from the Question model.
* The different model field types ([DateTimeField](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.DateTimeField), [CharField](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.CharField)) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
* [DateTimeField](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.DateTimeField) gets free JavaScript shortcuts. Dates get “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 change-list page.
* **Save and continue editing**: Saves changes & reloads admin page
* **Save and add another:** Saves changes & loads new, blank form
* **Delete** – Displays a delete confirmation page.

If the value of “Date published” doesn’t match the time when you created the question in [Tutorial 1](https://docs.djangoproject.com/en/2.2/intro/tutorial01/), it probably means you forgot to set the correct value for the [TIME\_ZONE](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TIME_ZONE) setting. Change it, reload the page and check that the correct value appears. 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:



When you’re comfortable with the models API and have familiarized yourself with the admin site, read [part 3 of this tutorial](https://docs.djangoproject.com/en/2.2/intro/tutorial03/) to learn about how to add more views to our polls app.

# your first Django app, part 3

This tutorial begins where [Tutorial 2](https://docs.djangoproject.com/en/2.2/intro/tutorial02/) left off. We will focus on creating the public interface – “views.”

## Overview

A view is a “type” of Web page in your Django application that generally serves a specific function and has a specific template. For example, in a blog application, you might have the following views:

* **Blog homepage** – displays the latest few entries.
* **Entry “detail” page** – permalink page for a single entry.
* **Year-based archive page** – displays all months with entries in the given year.
* **Month-based archive page** – displays all days with entries in the given month.
* **Day-based archive page** – displays all entries in the given day.
* **Comment action** – handles posting comments to a given entry.

In our poll application, we’ll have four views:

* **Question “index” page** – displays the latest few questions.
* **Question “detail” page** – displays a question text, with no results but with a form to vote.
* **Question “results” pag**e – displays results for a question.
* **Vote action** – handles voting for a particular choice in a particular question.

In Django, web pages and other content are delivered by views. Each view is represented by a simple Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that’s requested (to be precise, the part of the URL after the domain name). On the web you may have come across “ME2/Sites/dirmod.asp?sid=&type=gen&mod =Core+Pages&gid= A6CD4967199A42D9B65B1B”. Django allows us much more elegant *URL patterns* than that. A URL pattern is simply the general form of a URL: /newsarchive/<year>/<month>/. To get from a URL to a view, Django uses what are known as ‘URLconfs’. A URLconf maps URL patterns to views. This tutorial provides basic instruction in the use of URLconfs, and you can refer to [URL dispatcher](https://docs.djangoproject.com/en/2.2/topics/http/urls/) for more information.

## Writing more views

Now let’s add a few more views to polls/views.py. These views are slightly different, because they take an argument:

polls/views.py

def detail(request, question\_id):

return HttpResponse("You look at question %s." % question\_id)

def results(request, question\_id):

response = "You look at the results of question %s."

return HttpResponse(response % question\_id)

def vote(request, question\_id):

return HttpResponse("You vote on question %s." % question\_id)

Wire these new views into the polls.urls module by adding the following [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) calls:

polls/urls.py

from django.urls import path

from . import views

urlpatterns = [

# ex: /polls/

path('', views.index, name='index'),

# ex: /polls/5/

path('<int:question\_id>/', views.detail, name='detail'),

# ex: /polls/5/results/

path('<int:question\_id>/results/', views.results, name='results'),

# ex: /polls/5/vote/

path('<int:question\_id>/vote/', views.vote, name='vote'),

]

Look in your browser, at “/polls/34/”. It’ll run the detail() method and display the ID you provide in the URL. Try “/polls/34/results/” and “/polls/34/vote/”: these will display the placeholder results and voting pages. When a page is requested from your website – say, “/polls/34/”, Django will load the mysite.urls Python module because it’s pointed to by the [ROOT\_URLCONF](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-ROOT_URLCONF) setting. It finds the variable named urlpatterns and traverses the patterns in order. After finding the match at 'polls/', it strips off the matching text ("polls/") and sends the remaining text – "34/" – to the ‘polls.urls’ URLconf for further processing. There it matches '<int:question\_id>/', resulting in a call to the detail() view:

detail(request=<HttpRequest object>, question\_id=34)

The question\_id=34 part comes from <int:question\_id>. Using angle brackets “captures” part of the URL and sends it as a keyword argument to the view function. The :question\_id> part of the string defines the name that will be used to identify the matched pattern, and the <int: part is a converter that determines what patterns should match this part of the URL path. There’s no need to add URL cruft such as .html: something like this:

path('polls/latest.html', views.index),

## Write views that actually do something

Each view is responsible for: returning an [HttpResponse](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponse) object containing the content for the requested page, or raising an exception such as [Http404](https://docs.djangoproject.com/en/2.2/topics/http/views/#django.http.Http404). The rest is up to you. Your view can read records from a database, or use a template system such as Django’s – or a third-party Python template system, or generate a PDF file, output XML, create a ZIP file on the fly, etc. All Django wants is that [HttpResponse](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponse). Or an exception. For convenience, let’s use Django’s own database API, (covered in [Tutorial 2](https://docs.djangoproject.com/en/2.2/intro/tutorial02/)). Here’s one stab at a new index() view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

polls/views.py

from django.http import HttpResponse

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

output = ', '.join([q.question\_text for q in latest\_question\_list])

return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged

There’s a problem here, though: the page’s design is hard-coded in the view. If you want to change the way the page looks, you’ll have to edit this Python code. So let’s use Django’s template system to separate the design from Python by creating a template that the view can use. First, create a directory called templates in your polls directory. Django will look for templates in there. Your project’s [TEMPLATES](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES) setting describes how Django will load and render templates. The default settings file configures a DjangoTemplates backend whose [APP\_DIRS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES-APP_DIRS) option is set to True. By convention DjangoTemplates looks for a “templates” subdirectory in each of the [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS). Within the templates directory you have just created, create another directory called polls, and index.html: your template should be at polls/templates/polls/index.html. Because of how the app\_directories template loader works as described above, refer to this template within Django simply as polls/index.html.

### Template namespacing

We *could* put our templates directly in polls/templates (rather than creating another polls subdirectory). But Django will choose the first template it finds whose name matches. If you had a template with the same name in a *different* application, Django could not distinguish between them. We need to be able to point Django at the right one. The easiest way is by *namespacing* them - put those templates inside *another* directory named for the application. Put the following code in that template:

polls/templates/polls/index.html

{% if latest\_question\_list %}

<ul>

{% for question in latest\_question\_list %}

<li><a href="/polls/{{ question.id }}/">{{ question.question\_text }}</a></li>

{% endfor %}

</ul>

{% else %}

<p>No polls are available.</p>

{% endif %}

Update index view in polls/views.py to use the template:

polls/views.py

from django.http import HttpResponse

from django.template import loader

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

template = loader.get\_template('polls/index.html')

context = {

'latest\_question\_list': latest\_question\_list,

}

return HttpResponse(template.render(context, request))

That code loads the template polls/index.html and passes it a context. *The context is a dictionary mapping template variable names to Python objects*. Load the page by pointing your browser at “/polls/”. See a bulleted-list containing the “What’s up” question from [Tutorial 2](https://docs.djangoproject.com/en/2.2/intro/tutorial02/)? The link points to the question’s detail page.

### A shortcut: [render()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.render)

A common idiom is to load a template, fill a context and return an [HttpResponse](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponse) object with the result of the rendered template. Django provides a shortcut. Full index() view, rewritten:

polls/views.py

from django.shortcuts import render

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

context = {'latest\_question\_list': latest\_question\_list}

return render(request, 'polls/index.html', context)

Once we’ve done this in all these views, we won’t need to import [loader](https://docs.djangoproject.com/en/2.2/topics/templates/#module-django.template.loader) and [HttpResponse](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponse) (keep HttpResponse if you still have the stub methods for detail, results, and vote). The [render()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.render) function takes: request object as argument 1, template name as argument 2 and dictionary as (optional) argument 3. It returns an [HttpResponse](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponse) object of the given template rendered with the given context.

### Raising a 404 error

Now, let’s tackle the question detail view – the page that displays the question text for a given poll. Here’s the view:

polls/views.py

from django.http import Http404

from django.shortcuts import render

from .models import Question

# ...

def detail(request, question\_id):

try:

question = Question.objects.get(pk=question\_id)

except Question.DoesNotExist:

raise Http404("Question does not exist")

return render(request, 'polls/detail.html', {'question': question})

New concept: The view raises the [Http404](https://docs.djangoproject.com/en/2.2/topics/http/views/#django.http.Http404) exception if a question with the requested ID doesn’t exist. We’ll discuss what you could put in that polls/detail.html template a bit later, but if you’d like to quickly get the above example working, a file containing just:

polls/templates/polls/detail.html

{{ question }}

will get you started for now.

### A shortcut: [get\_object\_or\_404()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.get_object_or_404)

It’s a very common idiom to use [get()](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.get) and raise [Http404](https://docs.djangoproject.com/en/2.2/topics/http/views/#django.http.Http404) if the object doesn’t exist. Django provides a shortcut. Here’s the detail() view, rewritten:

polls/views.py

from django.shortcuts import get\_object\_or\_404, render

from .models import Question

# ...

def detail(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

return render(request, 'polls/detail.html', {'question': question})

The [get\_object\_or\_404()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.get_object_or_404) function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the [get()](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.get) function of the model’s manager. It raises [Http404](https://docs.djangoproject.com/en/2.2/topics/http/views/#django.http.Http404) if the object doesn’t exist.

### Philosophy

Why do we use a helper function [get\_object\_or\_404()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.get_object_or_404) instead of automatically catching the [ObjectDoesNotExist](https://docs.djangoproject.com/en/2.2/ref/exceptions/#django.core.exceptions.ObjectDoesNotExist) exceptions at a higher level, or having the model API raise [Http404](https://docs.djangoproject.com/en/2.2/topics/http/views/#django.http.Http404) instead of [ObjectDoesNotExist](https://docs.djangoproject.com/en/2.2/ref/exceptions/#django.core.exceptions.ObjectDoesNotExist)? Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling. Some controlled coupling is introduced in the [django.shortcuts](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#module-django.shortcuts) module.

There’s also a [get\_list\_or\_404()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.get_list_or_404) function, which works just as [get\_object\_or\_404()](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/#django.shortcuts.get_object_or_404) – except using [filter()](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.filter) instead of [get()](https://docs.djangoproject.com/en/2.2/ref/models/querysets/#django.db.models.query.QuerySet.get). It raises [Http404](https://docs.djangoproject.com/en/2.2/topics/http/views/#django.http.Http404) if the list is empty.

### Use the template system

Back to the detail() view. Given the context variable question, here’s what the polls/detail.html template might look like:

polls/templates/polls/detail.html

<h1>{{ question.question\_text }}</h1>

<ul>

{% for choice in question.choice\_set.all %}

<li>{{ choice.choice\_text }}</li>

{% endfor %}

</ul>

The template system uses dot-lookup syntax to access variable attributes. In the example of {{ question.question\_text }}, first Django does a dictionary lookup on the object question. Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup had failed, it would’ve tried a list-index lookup.

Method-calling happens in the [{% for %}](https://docs.djangoproject.com/en/2.2/ref/templates/builtins/#std:templatetag-for) loop: question.choice\_set.all is interpreted as the Python code question.choice\_set.all(), which returns an iterable of Choice objects and is suitable for use in the [{% for %}](https://docs.djangoproject.com/en/2.2/ref/templates/builtins/#std:templatetag-for) tag.

See the [template guide](https://docs.djangoproject.com/en/2.2/topics/templates/) for more about templates.

### Removing hardcoded URLs in templates

Remember, when we wrote the link to a question in the polls/index.html template, the link was partially hardcoded like this:

<li><a href="/polls/{{ question.id }}/">{{ question.question\_text }}</a></li>

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates. However, since you defined the name argument in the [path()](https://docs.djangoproject.com/en/2.2/ref/urls/#django.urls.path) functions in the polls.urls module, you can remove a reliance on specific URL paths defined in your url configurations by using the {% url %} template tag:

<li><a href="{% url 'detail' question.id %}">{{ question.question\_text }}</a></li>

The way this works is by looking up the URL definition as specified in the polls.urls module. You can see exactly where the URL name of ‘detail’ is defined below:

...

# the 'name' value as called by {% url %} template tag

path('<int:question\_id>/', views.detail, name='detail'),

...

If you want to change the URL of the polls detail view to something else, perhaps to something like polls/specifics/12/ instead of doing it in the template (or templates) you would change it in polls/urls.py:

...

# added the word 'specifics'

path('specifics/<int:question\_id>/', views.detail, name='detail'),

...

### Namespacing URL names

Real Django projects might have 5 apps or more. How does Django differentiate their URL names? Example: the polls app has a detail view, and so might an app on the same project that is for a blog. How can Django know which app view to create for a url when using the {% url %} template tag? Answer: add namespaces to your URLconf. In polls/urls.py, add app\_name to set namespace:

polls/urls.py

from django.urls import path

from . import views

app\_name = 'polls'

urlpatterns = [

path('', views.index, name='index'),

path('<int:question\_id>/', views.detail, name='detail'),

path('<int:question\_id>/results/', views.results, name='results'),

path('<int:question\_id>/vote/', views.vote, name='vote'),

]

Now change your polls/index.html template from:

polls/templates/polls/index.html

<li><a href="{% url 'detail' question.id %}">{{ question.question\_text }}</a></li>

to point at the namespaced detail view:

polls/templates/polls/index.html

<li><a href="{% url '**polls:detail**' question.id %}">{{ question.question\_text }}</a></li>

When you’re comfortable with writing views, read [part 4 of this tutorial](https://docs.djangoproject.com/en/2.2/intro/tutorial04/) to learn about simple form processing and generic views.

# your first Django app, part 4

This tutorial follows [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/). We continue the Web-poll to focus on simple form processing and cutting down our code.

## Write a simple form

Let’s update our poll detail template (“polls/detail.html”), so that the template contains an HTML <form> element:

polls/templates/polls/detail.html

<h1>{{ question.question\_text }}</h1>

{% if error\_message %}<p><strong>{{ error\_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">

{% csrf\_token %}

{% for choice in question.choice\_set.all %}

<input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}">

<label for="choice{{ forloop.counter }}">{{ choice.choice\_text }}</label><br>

{% endfor %}

<input type="submit" value="Vote">

</form>

A quick rundown:

* The template displays a radio button for each question. The value of each radio button is the associated question choice’s ID. The name of each radio button is "choice". Select a radio button and submit the form; a POST data choice=# is sent where # is the ID of the selected choice. This is the basic concept of HTML forms.
* The form’s action is set to {% url 'polls:vote' question.id %}, and method="post". Notice method="post" is used, not method="get"! When creating a form that alters data server-side, use method="post" (not specific to Django good Web practice).
* forloop.counter indicates how many times the [for](https://docs.djangoproject.com/en/2.2/ref/templates/builtins/#std:templatetag-for) tag has gone through its loop
* Since a POST form can modify data, we need to worry about **Cross Site Request Forgeries**. In Django, to protect against it, all POST forms that are targeted at internal URLs should use the [{% csrf\_token %}](https://docs.djangoproject.com/en/2.2/ref/templates/builtins/#std:templatetag-csrf_token) template tag.

Now, let’s create a Django view that handles the submitted data and does something with it. Remember, in [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/), we created a URLconf for the polls application that includes this line:

polls/urls.py

path('<int:question\_id>/vote/', views.vote, name='vote'),

We also created a dummy implementation of the vote() function. Let’s create a real version. Add the following to polls/views.py:

polls/views.py

from django.http import HttpResponse, HttpResponseRedirect

from django.shortcuts import get\_object\_or\_404, render

from django.urls import reverse

from .models import Choice, Question

# ...

def vote(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

try:

selected\_choice = question.choice\_set.get(pk=request.POST ['choice'])

except (KeyError, Choice.DoesNotExist):

# Redisplay the question voting form.

return render(request, 'polls/detail.html', {

'question': question,

'error\_message': "You didn't select a choice.",

})

else:

selected\_choice.votes += 1

selected\_choice.save()

# Always return an HttpResponseRedirect after

# successfully dealing with POST data. This

# prevents data from being posted twice if a

# user hits the Back button.

return HttpResponseRedirect(reverse ('polls:results', args=(question.id,)))

New in this code:

* [request.POST](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest.POST) is a dictionary-like object that lets you access submitted data by key name: request.POST['choice'] returns the selected choice ID as a string ([request.POST](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest.POST) values are always strings). Django provides [request.GET](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest.GET) for accessing GET data in the same way. But we explicitly use [request.POST](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest.POST) to ensure that data is only altered via a POST call.
* request.POST['choice'] raises [KeyError](https://docs.python.org/3/library/exceptions.html#KeyError) if choice not provided in POST data. The above checks for [KeyError](https://docs.python.org/3/library/exceptions.html#KeyError) and redisplays question form with error message if choice not given.
* After incrementing the choice count, rather than a normal [HttpResponse](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponse), the code returns [HttpResponseRedirect](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponseRedirect), which takes 1 argument: the URL to which the user will be redirected (see next point for how we construct the URL). Always return an [HttpResponseRedirect](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponseRedirect) after successfully dealing with POST data (not specific to Django).
* We use reverse() in [HttpResponseRedirect](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpResponseRedirect) constructor. This function helps avoid hardcoding URL in view. It gets the name of the view to pass control to + the variable portion of the URL pointing to that view. Using URLconf set up in [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/), [reverse()](https://docs.djangoproject.com/en/2.2/ref/urlresolvers/#django.urls.reverse) call returns a string like: '/polls/3/results/' where ‘3’ = value of question.id. This redirected URL then calls 'results' view to display the final page.

As mentioned in [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/), request is an [HttpRequest](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest) object. For more on [HttpRequest](https://docs.djangoproject.com/en/2.2/ref/request-response/#django.http.HttpRequest) objects, see [request and response documentation](https://docs.djangoproject.com/en/2.2/ref/request-response/). After somebody votes in a question, the vote() view redirects to the results page for the question.

polls/views.py

from django.shortcuts import get\_object\_or\_404, render

def results(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

return render(request, 'polls/results.html', {'question': question})

This is almost exactly the same as the detail() view from [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/). The only difference is the template name. We’ll fix this redundancy later. Now, create a polls/results.html template:

polls/templates/polls/results.html

<h1>{{ question.question\_text }}</h1>

<ul>

{% for choice in question.choice\_set.all %}

<li>{{ choice.choice\_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>

{% endfor %}

</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>

Now, go to /polls/1/ in browser and vote in question. You’ll see a results page updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

Note: The code for vote() view has a small problem. It first gets the selected\_choice object from the database, then computes the new value of votes, and then saves it back to the database. If two users of your website try to vote at *exactly the same time*, this might go wrong: The same value, let’s say 42, will be retrieved for votes. Then, for both users the new value of 43 is computed and saved, but 44 would be the expected value. This is called a *race condition*. If you are interested, read [Avoiding race conditions using F()](https://docs.djangoproject.com/en/2.2/ref/models/expressions/#avoiding-race-conditions-using-f).

## Use generic views: Less code is better

The detail() (from [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/)) and results() views are simple – and redundant. So is index() view, which displays a list of polls. A common case of Web development: get data from database based on a parameter passed in the URL, load a template and return the rendered template. Because it is so common, Django provides a shortcut, called the “generic views” system, which abstract common patterns to the point where Python code is not necessary. Let’s convert our poll app to use the generic views system, so we can delete a bunch of our own code. We’ll just have to take a few steps to make the conversion. We will:

1. Convert the URLconf.
2. Delete some of the old, unneeded views.
3. Introduce new views based on Django’s generic views.

Why the code-shuffle? Generally, you should evaluate whether generic views are a good fit, then use them from start, rather than refactoring the code halfway through. But this tutorial intentionally focused on writing the views “the hard way” until now.

### Amend URLconf

First, open the polls/urls.py URLconf and change:

polls/urls.py

from django.urls import path

from . import views

app\_name = 'polls'

urlpatterns = [

path('', views.IndexView.as\_view(), name='index'),

path('<int:pk>/', views.DetailView.as\_view(), name='detail'),

path('<int:pk>/results/', views.ResultsView.as\_view(), name='results'),

path('<int:question\_id>/vote/', views.vote, name='vote'),

]

Note that the name of the matched pattern in the path strings of the 2nd and 3rd patterns has changed from <question\_id> to <pk>.

### Amend views

Next, we’re going to remove our old index, detail, and results views and use Django’s generic views instead. To do so, open the polls/views.py file and change it like so:

polls/views.py

from django.http import HttpResponseRedirect

from django.shortcuts import get\_object\_or\_404, render

from django.urls import reverse

from django.views import generic

from .models import Choice, Question

class IndexView(generic.ListView):

template\_name = 'polls/index.html'

context\_object\_name = 'latest\_question\_list'

def get\_queryset(self):

"""Return the last five published questions."""

return Question.objects.order\_by('-pub\_date')[:5]

class DetailView(generic.DetailView):

model = Question

template\_name = 'polls/detail.html'

class ResultsView(generic.DetailView):

model = Question

template\_name = 'polls/results.html'

def vote(request, question\_id):

... # same as above, no changes needed.

We use 2 generic views: [ListView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.list.ListView) and [DetailView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView) to respectively abstract the concepts of “display a list of objects” and “display a detail page for a particular type of object.”

* Each generic view needs to know what model it will be acting upon. This is provided using the model attribute.
* The [DetailView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView) generic view expects the primary key value captured from the URL to be called "pk", so we’ve changed question\_id to pk for the generic views.

By default, the [DetailView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView) generic view uses the template: <app name>/<model name>\_detail.html. Here, it uses template "polls/question\_detail.html". The template\_name attribute is used to tell Django to use a specific template name instead of the autogenerated default template name. We also specify the template\_name for the results list view: this ensures that the results and detail view have a different appearance, even though both are [DetailView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView). Similarly, [ListView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.list.ListView) generic view uses default template <app name>/<model name>\_list.html; using template\_name tells [ListView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.list.ListView) to use "polls/index.html" template. Previously, the templates contain the question and latest\_question\_list context variables. For DetailView the question variable is provided automatically, since we use Django model (Question). However, for ListView, the automatically generated context variable is question\_list. To override, we provide the context\_object\_name attribute, specifying that we want to use latest\_question\_list instead. Alternatively, change your templates to match the new default context variables (but it’s easier to tell Django to use the variable you want). Run the server, and use your new polling app based on generic views. For full details on generic views, see the [generic views documentation](https://docs.djangoproject.com/en/2.2/topics/class-based-views/). When you’re comfortable with forms and generic views, read [part 5 of this tutorial](https://docs.djangoproject.com/en/2.2/intro/tutorial05/) to learn about testing our polls app.

# your first Django app, part 5

This tutorial begins where [Tutorial 4](https://docs.djangoproject.com/en/2.2/intro/tutorial04/) left off. We’ve built a Web-poll application, and we’ll now create some automated tests for it.

## Introducing automated testing

### What are automated tests?

Tests are simple routines that check the operation of your code. Some tests might apply to a tiny detail (*does a particular model method return values as expected?*) while others examine the overall operation of the software (*does a sequence of user inputs on the site produce the desired result?*). That’s no different from the kind of testing you did earlier in [Tutorial 2](https://docs.djangoproject.com/en/2.2/intro/tutorial02/), using the [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell) to examine the behavior of a method, or running the application and entering data to check how it behaves. What’s different in *automated* tests is that the testing work is done for you by the system. You create a set of tests once, and then as you make changes to your app, you can check that your code still works as you originally intended, without time consuming manual testing.

### Why you need to create tests

So why create tests, and why now? You may feel that you have quite enough on your plate just learning Python/Django, and having yet another thing to learn and do may seem overwhelming and perhaps unnecessary. After all, our polls application is working quite happily now; going through the trouble of creating automated tests is not going to make it work any better. If creating the polls application is the last bit of Django programming you will ever do, then true, you don’t need to know how to create automated tests. But, if that’s not the case, now is an excellent time to learn.

### Tests will save you time

Up to a certain point, ‘checking that it seems to work’ will be a satisfactory test. In a more sophisticated application, you might have dozens of complex interactions between components. A change in any of those components could have unexpected consequences on the application’s behavior. Checking that it still ‘seems to work’ could mean running through your code’s functionality with twenty different variations of your test data just to make sure you haven’t broken something - not a good use of your time. That’s especially true when automated tests could do this for you in seconds. If something’s gone wrong, tests will also assist in identifying the code that’s causing the unexpected behavior. Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly. However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

### Tests don’t just identify problems, they prevent them

It’s a mistake to think of tests merely as a negative aspect of development. Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it’s your own code, you will sometimes find yourself poking around in it trying to find out what exactly it’s doing. Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - *even if you hadn’t even realized it had gone wrong*.

### Tests make your code more attractive

You might have created a brilliant piece of software, but you will find that many other developers will simply refuse to look at it because it lacks tests; without tests, they won’t trust it. Jacob Kaplan-Moss, one of Django’s original developers, says “Code without tests is broken by design.” That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.

### Tests help teams work together

The previous points are written from the point of view of a single developer maintaining an application. Complex applications will be maintained by teams. Tests guarantee that colleagues don’t inadvertently break your code (and that you don’t break theirs without knowing). If you want to make a living as a Django programmer, you must be good at writing tests!

### Basic testing strategies

Many ways to approach writing tests. Some programmers follow a discipline called “[test-driven development](https://en.wikipedia.org/wiki/Test-driven_development)”; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it’s similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development simply formalizes the problem in a Python test case. More often, a newcomer to testing will create some code and later decide that it should have some tests. Perhaps it would have been better to write some tests earlier, but it’s never too late to get started. Sometimes it’s difficult to figure out where to get started with writing tests. If you have written several thousand lines of Python, choosing something to test might not be easy. In such a case, it’s fruitful to write your first test the next time you make a change, either when you add a new feature or fix a bug.

## Writing our first test

### We identify a bug

Question.was\_published\_recently() returns True if the Question was published within the last day (which is correct) but also if the Question’s pub\_date field is in the future (which certainly isn’t). Confirm the bug by using the [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell) to check the method on a question whose date lies in the future:

$ python manage.py shell

>>> import datetime

>>> from django.utils import timezone

>>> from polls.models import Question

>>> # create a Question instance with pub\_date 30 days in the future

>>> future\_question = Question(pub\_date=timezone.now() + datetime.timedelta(days=30))

>>> # was it published recently?

>>> future\_question.was\_published\_recently()

True

Since things in the future are not ‘recent’, this is clearly wrong.

### Create a test to expose the bug

What we’ve just done in the [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell) to test for the problem is exactly what we can do in an automated test, so let’s turn that into an automated test. A conventional place for an application’s tests is in the application’s tests.py file; the testing system will automatically find tests in any file whose name begins with test. Put the following in the tests.py file in the polls application:

polls/tests.py

import datetime

from django.test import TestCase

from django.utils import timezone

from .models import Question

class QuestionModelTests(TestCase):

def test\_was\_published\_recently\_with\_future\_question (self):

"""

was\_published\_recently(): returns False for questions whose pub\_date is in the future.

"""

time = timezone.now() + datetime.timedelta(days=30)

future\_question = Question(pub\_date=time)

self.assertIs (future\_question.was\_published\_recently(), False)

Here we created [django.test.TestCase](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.TestCase) subclass with a method that creates a Question instance with a pub\_date in the future. We check the output of was\_published\_recently() (*ought* to be False).

### Running tests

In the terminal, we can run our test:

$ python manage.py test polls

and you’ll see something like:

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

F

========================================================

FAIL: test\_was\_published\_recently\_with\_future\_question (polls.tests.QuestionModelTests)

--------------------------------------------------------

Traceback (most recent call last):

File "/path/to/mysite/polls/tests.py", line 16, in test\_was\_published\_recently\_with\_future\_question

self.assertIs(future\_question. was\_published\_recently(), False)

AssertionError: True is not False

--------------------------------------------------------

Ran 1 test in 0.001s

FAILED (failures=1)

Destroying test database for alias 'default'...

What happened:

* manage.py test polls looked for tests in the polls application
* it found a subclass of the [django.test.TestCase](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.TestCase) class
* it created a special database for the purpose of testing
* it looked for test methods - ones whose names begin with test
* in test\_was\_published\_recently\_with\_future\_question it created a Question instance whose pub\_date field is 30 days in the future
* using assertIs(), it discovered that was\_published\_recently() returns True, though we wanted it to return False

The test informs us which test failed and even the line on which the failure occurred.

### Fixing the bug

We know the problem: Question.was\_published\_recently() to return False if pub\_date is in the future. Amend the method in models.py, so that it only returns True if the date is also in the past:

polls/models.py

def was\_published\_recently(self):

now = timezone.now()

return now - datetime.timedelta(days=1) <= self.pub\_date <= now

and run the test again:

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.

---------------------------------------------------------

Ran 1 test in 0.001s

OK

Destroying test database for alias 'default'...

After identifying a bug, we wrote a test that exposes it and corrected the bug in the code so our test passes. Many other things might go wrong with our application in the future, but we can be sure that we won’t inadvertently reintroduce this bug, because simply running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

### More comprehensive tests

Let’s pin down was\_published\_recently() method; it would be embarrassing if in fixing one bug we had introduced another. Add two more test methods to the same class, to test the behavior of the method more comprehensively:

polls/tests.py

def test\_was\_published\_recently\_with\_old\_question(self):

"""

was\_published\_recently() returns False for questions

whose pub\_date is older than 1 day.

"""

time = timezone.now() - datetime.timedelta(days=1, seconds=1)

old\_question = Question(pub\_date=time)

self.assertIs(old\_question.was\_published\_recently(), False)

def test\_was\_published\_recently\_with\_recent\_question (self):

"""

was\_published\_recently() returns True for questions

whose pub\_date is within the last day.

"""

time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)

recent\_question = Question(pub\_date=time)

self.assertIs(recent\_question. was\_published\_recently(), True)

Now 3 tests to confirm that Question.was\_published\_recently() returns sensible values for past, recent, and future questions. Polls is now a simple application; but however complex it grows in the future and whatever other code it interacts with, we have guarantee that the method with written tests will behave in expected ways.

### Test a view

The polls application is undiscriminating: it publishes any question, including ones whose pub\_date field lies in the future. We should improve this. Setting a pub\_date in the future should mean that the Question is published at that moment, but invisible until then.

### A test for a view

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was a simple example of test-driven development, but it doesn’t really matter in which order we do the work. In our first test, we focused closely on the internal behavior of the code. For this test, we want to check its behavior as it would be experienced by a user through a web browser. Before we try to fix anything, let’s have a look at the tools at our disposal.

## The Django test client

Django provides a test [Client](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.Client) to simulate a user interacting with the code at the view level. We can use it in tests.py or even in the [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell).

We will start with the [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell) by setting up test environment in [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell):

$ python manage.py shell

>>> from django.test.utils import setup\_test\_environment

>>> setup\_test\_environment()

[setup\_test\_environment()](https://docs.djangoproject.com/en/2.2/topics/testing/advanced/#django.test.utils.setup_test_environment) installs a template renderer. This lets us examine additional attributes on responses such as response.context that otherwise wouldn’t be available. Note that this method *does not* setup a test database: the following is run against existing database; the output may differ slightly depending on what questions you created. You might get unexpected results if your TIME\_ZONE in settings.py isn’t correct. If you don’t remember setting it earlier, check it before continuing. Next we need to import the test client class (later in tests.py we will use the [django.test.TestCase](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.TestCase) class, which comes with its own client, so this won’t be required):

>>> from django.test import Client

>>> # create an instance of the client for our use

>>> client = Client()

With that ready, we can ask the client to do some work for us:

>>> # get a response from '/'

>>> response = client.get('/')

Not Found: /

>>> # we should expect a 404 from that address; if you

# instead see an "Invalid HTTP\_HOST header" error and

# a 400 response, you probably omitted the

# setup\_test\_environment() call described earlier.

>>> response.status\_code

404

>>> # on the other hand we should expect to find

# something at '/polls/' we'll use 'reverse()'

# rather than a hardcoded URL

>>> from django.urls import reverse

>>> response = client.get(reverse('polls:index'))

>>> response.status\_code

200

>>> response.content

b'\n <ul>\n \n <li><a href="/polls/1/">What&#39; sup?</a></li>\n \n </ul>\n\n'

>>> response.context['latest\_question\_list']

<QuerySet [<Question: What's up?>]>

### Improving our view

The list of polls shows polls that aren’t published yet (i.e. those that have a pub\_date in the future). Let’s fix that. In [Tutorial 4](https://docs.djangoproject.com/en/2.2/intro/tutorial04/) we introduced a class-based view, based on [ListView](https://docs.djangoproject.com/en/2.2/ref/class-based-views/generic-display/#django.views.generic.list.ListView):

polls/views.py

class IndexView(generic.ListView):

template\_name = 'polls/index.html'

context\_object\_name = 'latest\_question\_list'

def get\_queryset(self):

"""Return the last five published questions."""

return Question.objects.order\_by('-pub\_date')[:5]

Amend get\_queryset()so that it also checks the date by comparing it with timezone.now(). First we need to add an import:

polls/views.py

from django.utils import timezone

and then we must amend the get\_queryset method like so:

polls/views.py

def get\_queryset(self):

"""

Return the last five published questions (not

including those set to be published in the future).

"""

return Question.objects.filter(pub\_date\_\_lte =timezone.now()).order\_by('-pub\_date')[:5]

Question.objects.filter(pub\_date\_\_lte=timezone.now()) returns a queryset containing Questions whose pub\_date is less than or equal to - that is, earlier than or equal to - timezone.now.

### Testing our new view

Now you can satisfy yourself that this behaves as expected by firing up runserver, loading the site in your browser, creating Questions with dates in the past and future, and checking that only those that have been published are listed. You don’t want to have to do that *every single time you make any change that might affect this* - so let’s also create a test, based on our [shell](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-shell) session above.

Add the following to polls/tests.py:

polls/tests.py

from django.urls import reverse

Create a shortcut function to create questions and a new test class:

polls/tests.py

def create\_question(question\_text, days):

"""

Create a question with the given `question\_text` and

published the given number of `days` offset to now

(negative for questions published past, positive for

questions that have yet to be published).

"""

time = timezone.now() + datetime.timedelta(days=days)

return Question.objects.create( question\_text=question\_text, pub\_date=time)

class QuestionIndexViewTests(TestCase):

def test\_no\_questions(self):

"""

If no questions exist, an appropriate message is

displayed.

"""

response = self.client.get(reverse('polls:index'))

self.assertEqual(response.status\_code, 200)

self.assertContains(response, "No polls are available.")

self.assertQuerysetEqual(response.context ['latest\_question\_list'], [])

def test\_past\_question(self):

"""

Questions with a pub\_date in the past are

displayed on the index page.

"""

create\_question(question\_text="Past question.", days=-30)

response = self.client.get( reverse('polls:index'))

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question.>']

)

def test\_future\_question(self):

"""

Questions with a pub\_date in the future aren't

displayed on the index page.

"""

create\_question(question\_text="Future question.", days=30)

response = self.client.get(reverse('polls:index'))

self.assertContains(response, "No polls are available.")

self.assertQuerysetEqual(response.context[ 'latest\_question\_list'], [])

def test\_future\_question\_and\_past\_question(self):

"""

Even if both past and future questions exist,

only past questions are displayed.

"""

create\_question(question\_text="Past question.", days=-30)

create\_question(question\_text="Future question.", days=30)

response = self.client.get(reverse('polls:index'))

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question.>']

)

def test\_two\_past\_questions(self):

"""

The questions index page may display multiple

questions.

"""

create\_question(question\_text="Past question 1.", days=-30)

create\_question(question\_text="Past question 2.", days=-5)

response = self.client.get(reverse('polls:index'))

self.assertQuerysetEqual(

response.context['latest\_question\_list'],

['<Question: Past question 2.>', '<Question: Past question 1.>']

)

create\_question() takes some repetition out of the process of creating questions. test\_no\_questions() doesn’t create any questions, but checks the message: “No polls are available.” and verifies the latest\_question\_list is empty. Note that [django.test.TestCase](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.TestCase) class provides some additional assertion methods: [assertContains()](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.SimpleTestCase.assertContains) and [assertQuerysetEqual()](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.TransactionTestCase.assertQuerysetEqual). In test\_past\_question(), we create a question and verify that it appears in the list. In test\_future\_question, we create a question with a pub\_date in the future. The database is reset for each test method, so the first question is no longer there, and so again the index shouldn’t have any questions in it. And so on: we are using the tests to tell a story of admin input and user experience on the site, and checking that at every state and for every new change in the state of the system, the expected results are published.

### Testing the DetailView

Even though future questions don’t appear in the *index*, users can still reach them if they know or guess the right URL. So we need to add a similar constraint to DetailView:

polls/views.py

class DetailView(generic.DetailView):

...

def get\_queryset(self):

"""

Excludes any questions that aren't published yet.

"""

return Question.objects.filter(pub\_date\_\_lte =timezone.now())

And of course, we will add some tests, to check that a Question whose pub\_date is in the past can be displayed, and that one with a pub\_date in the future is not:

polls/tests.py

class QuestionDetailViewTests(TestCase):

def test\_future\_question(self):

"""

The detail view of a question with a pub\_date in

the future returns a 404 not found.

"""

future\_question = create\_question(question\_text='Future question.', days=5)

url = reverse('polls:detail', args=(future\_question.id,))

response = self.client.get(url)

self.assertEqual(response.status\_code, 404)

def test\_past\_question(self):

"""

The detail view of a question with a pub\_date in

the past displays the question's text.

"""

past\_question = create\_question(question\_text='Past Question.', days=-5)

url = reverse('polls:detail', args=(past\_question.id,))

response = self.client.get(url)

self.assertContains(response, past\_question.question\_text)

### Ideas for more tests

We ought to add a similar get\_queryset method to ResultsView and create a new test class for that view. It’ll be very similar and a lot of repetition. We could also improve our application in other ways, adding tests along the way. For example, it’s silly that Questions can be published on the site that have no Choices. So, our views could check for this, and exclude such Questions. Our tests would create a Question without Choices and then test that it’s not published, as well as create a similar Question *with* Choices, and test that it *is* published. Perhaps logged-in admin users should be allowed to see unpublished Questions, but not ordinary visitors. Again: whatever needs to be added to the software to accomplish this should be accompanied by a test, whether you write the test first and then make the code pass the test, or work out the logic in your code first and then write a test to prove it. At a certain point you are bound to look at your tests and wonder whether your code is suffering from test bloat, which brings us to:

### When testing, more is better

Our tests seem growing out of control. There will be more code in our tests than in our application, and the repetition unaesthetic, compared to the elegant conciseness of the rest of our code.

**It doesn’t matter**. Let them grow. You can write a test once and then forget about it. It will continue performing its useful function as you continue to develop your program. Sometimes tests will need to be updated. Suppose that we amend our views so that only Questions with Choices are published. In that case, many of our existing tests will fail - *telling us exactly which tests need to be amended to bring them up to date*, so to that extent tests help look after themselves. At worst, as you continue developing, you might find that you have some tests that are now redundant. Even that’s not a problem; in testing redundancy is a *good* thing. As long as your tests are sensibly arranged, they won’t become unmanageable. Good rules-of-thumb include having:

* a separate TestClass for each model or view
* a separate test method for each set of conditions you want to test
* test method names that describe their function

## Further testing

This tutorial only introduces some of the basics of testing. There’s a great deal more you can do, and a number of very useful tools at your disposal to achieve some very clever things. For example, while our tests here have covered some of the internal logic of a model and the way our views publish information, you can use an “in-browser” framework such as [Selenium](http://seleniumhq.org/) to test the way your HTML actually renders in a browser. These tools allow you to check not just the behavior of your Django code, but also, for example, of your JavaScript. It’s quite something to see the tests launch a browser, and start interacting with your site, as if a human being were driving it! Django includes [LiveServerTestCase](https://docs.djangoproject.com/en/2.2/topics/testing/tools/#django.test.LiveServerTestCase) to facilitate integration with tools like Selenium. If you have a complex application, you may want to run tests automatically with every commit for the purposes of [continuous integration](https://en.wikipedia.org/wiki/Continuous_integration), so that quality control is itself - at least partially - automated. A good way to spot untested parts of your application is to check code coverage. This also helps identify fragile or even dead code. If you can’t test a piece of code, it usually means that code should be refactored or removed. Coverage will help to identify dead code. See [Integration with coverage.py](https://docs.djangoproject.com/en/2.2/topics/testing/advanced/#topics-testing-code-coverage) for details. [Testing in Django](https://docs.djangoproject.com/en/2.2/topics/testing/) has comprehensive information about testing.

## What’s next?

For full details on testing, see [Testing in Django](https://docs.djangoproject.com/en/2.2/topics/testing/). When you’re comfortable with testing Django views, read [part 6 of this tutorial](https://docs.djangoproject.com/en/2.2/intro/tutorial06/) to learn about static files management.

# your first Django app, part 6

This tutorial begins where [Tutorial 5](https://docs.djangoproject.com/en/2.2/intro/tutorial05/) left off. We’ve built a tested Web-poll application, and we’ll now add a stylesheet and an image. Aside from the HTML generated by the server, web applications generally need to serve additional files: images, JavaScript, or CSS, etc. to render the complete web page. In Django, we refer to these files as “*static files*”. For small projects, you can keep the static files somewhere your web server can find it. In bigger projects (multiple apps), dealing with multiple sets of static files provided by each application is tricky. That’s what django.contrib.staticfiles is for: it collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

## Customize your *app’s* look and feel

First, create directory static in your polls directory. Django will look for static files there, similarly to how Django finds templates inside polls/templates/. Django’s [STATICFILES\_FINDERS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-STATICFILES_FINDERS) setting contains a list of finders that know how to discover static files. One defaults is AppDirectoriesFinder, which looks for a “static” subdirectory in the [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS), like the one in polls we just created. The admin site uses the same directory structure for its static files. Within the static directory you have just created, create another directory called polls and within that create file style.css – i.e. your stylesheet polls/static/polls/style.css. Because of how the AppDirectoriesFinder staticfile finder works, refer to this static file in Django simply as polls/style.css, similar to how you reference the path for templates.

### Static file namespacing

Just like templates, we *could* get away with putting our static files directly in polls/static (rather than creating another polls subdirectory). But it would be a bad idea. Django chooses the first static file it finds whose name matches; if you had a static file with the same name in a *different* application, Django cannot distinguish between them. We need to point Django at the right one. The easiest way is by *namespacing*: putting those static files inside *another* directory named for the application itself. Put the following code in that stylesheet (polls/static/polls/style.css):

polls/static/polls/style.css

li a {

color: green;

}

Next, add at the top of polls/templates/polls/index.html:

polls/templates/polls/index.html

{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}">

The {% static %} template tag generates the absolute URL of static files. That’s all you need to do for development. Start the server (or restart it if it’s already running):

$ python manage.py runserver

Reload http://localhost:8000/polls/ and you should see that the question links are green (Django style!) which means that your stylesheet was properly loaded.

## Adding a background-image

Next, create a subdirectory for images. Create images subdirectory in polls/static/polls/ directory. Inside this directory, put an image called background.gif i.e. put image in: polls/static/polls/images/background.gif.

Then, add to stylesheet (polls/static/polls/style.css):

polls/static/polls/style.css

body {

background: white url("images/background.gif") no-repeat;

}

Reload http://localhost:8000/polls/. You see the background loaded in the top left of the screen.

**Warning**: Of course the {% static %} template tag is not available for use in static files like your stylesheet which aren’t generated by Django. Always use **relative paths** to link your static files between each other, because then you can change [STATIC\_URL](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-STATIC_URL) (used by the [static](https://docs.djangoproject.com/en/2.2/ref/templates/builtins/#std:templatetag-static) template tag to generate its URLs) without having to modify a bunch of paths in your static files as well. These are the **basics**. For more details on settings and other bits included with the framework see [the static files howto](https://docs.djangoproject.com/en/2.2/howto/static-files/) and [the staticfiles reference](https://docs.djangoproject.com/en/2.2/ref/contrib/staticfiles/). [Deploying static files](https://docs.djangoproject.com/en/2.2/howto/static-files/deployment/) discusses how to use static files on a real server. Read [part 7 of this tutorial](https://docs.djangoproject.com/en/2.2/intro/tutorial07/) to learn how to customize Django’s automatically-generated admin site.

# your first Django app, part 7

This tutorial continues [Tutorial 6](https://docs.djangoproject.com/en/2.2/intro/tutorial06/). We’re continuing the Web-poll application and will focus on customizing Django’s automatically-generated admin site that we first explored in [Tutorial 2](https://docs.djangoproject.com/en/2.2/intro/tutorial02/).

## Customize the admin form

Registering Question with admin.site.register(Question), Django can construct a default form representation. You’ll want to customize how the admin form looks and works. Do this by telling Django the options you want when you register the object. Let’s see how this works by reordering the fields on the edit form. Replace admin.site.register(Question) line with:

polls/admin.py

from django.contrib import admin

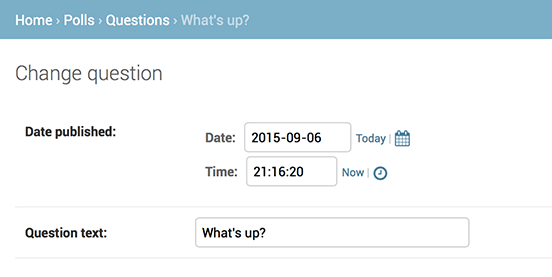
from .models import Question

class QuestionAdmin(admin.ModelAdmin):

fields = ['pub\_date', 'question\_text']

admin.site.register(Question, QuestionAdmin)

Follow this pattern – create a model admin class, then pass it as the argument 2 to admin.site.register() – when you need to change the admin options for a model. This particular change above makes the “Publication date” come before the “Question” field:



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:

polls/admin.py

from django.contrib import admin

from .models import Question

class QuestionAdmin(admin.ModelAdmin):

fieldsets = [

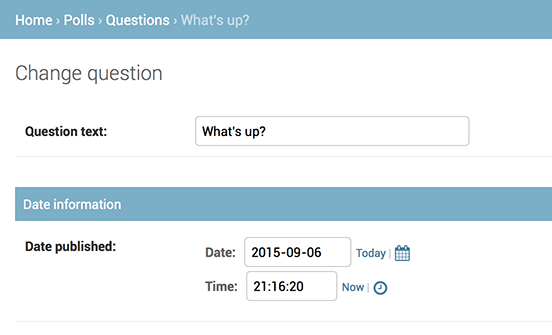
None, {'fields': ['question\_text']}),

('Date information',{'fields': ['pub\_date']}),

]

admin.site.register(Question, QuestionAdmin)

The first element of each tuple in [fieldsets](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.fieldsets) is the title of the fieldset. Here’s what our form looks like now:



## Adding related objects

OK, we have our Question admin page, but a Question has multiple Choices, and the admin page doesn’t display choices. Yet. There are two ways to solve this problem. The first is to register Choice with the admin just as we did with Question. That’s easy:

polls/admin.py

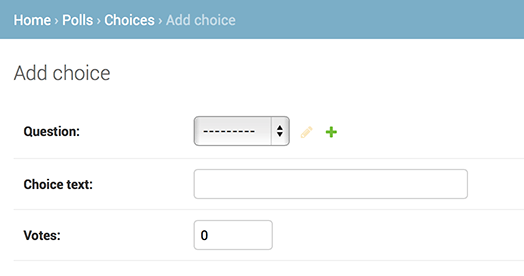
from django.contrib import admin

from .models import Choice, Question

# ...

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 “Question” field is a select box containing every question in the database. Django knows that a [ForeignKey](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.ForeignKey) should be represented in the admin as a <select> box. In our case, only one question exists at this point. Note the “Add Another” link next to “Question.” Every object with a ForeignKey relationship to another gets this for free. When click “Add Another”, you’ll get a popup window with the “Add question” form. If you add a question in that window and click “Save”, Django will save the question to the database and dynamically add it as the selected choice on the “Add choice” form you’re looking at. But, 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 Question object. Let’s make that happen. Remove register() call for the Choice model. Then, edit the Question registration code to read:

polls/admin.py

from django.contrib import admin

from .models import Choice, Question

class ChoiceInline(admin.StackedInline):

model = Choice

extra = 3

class QuestionAdmin(admin.ModelAdmin):

fieldsets = [

(None, {'fields': ['question\_text']}),

('Date information',{'fields': ['pub\_date'], 'classes': ['collapse']}),

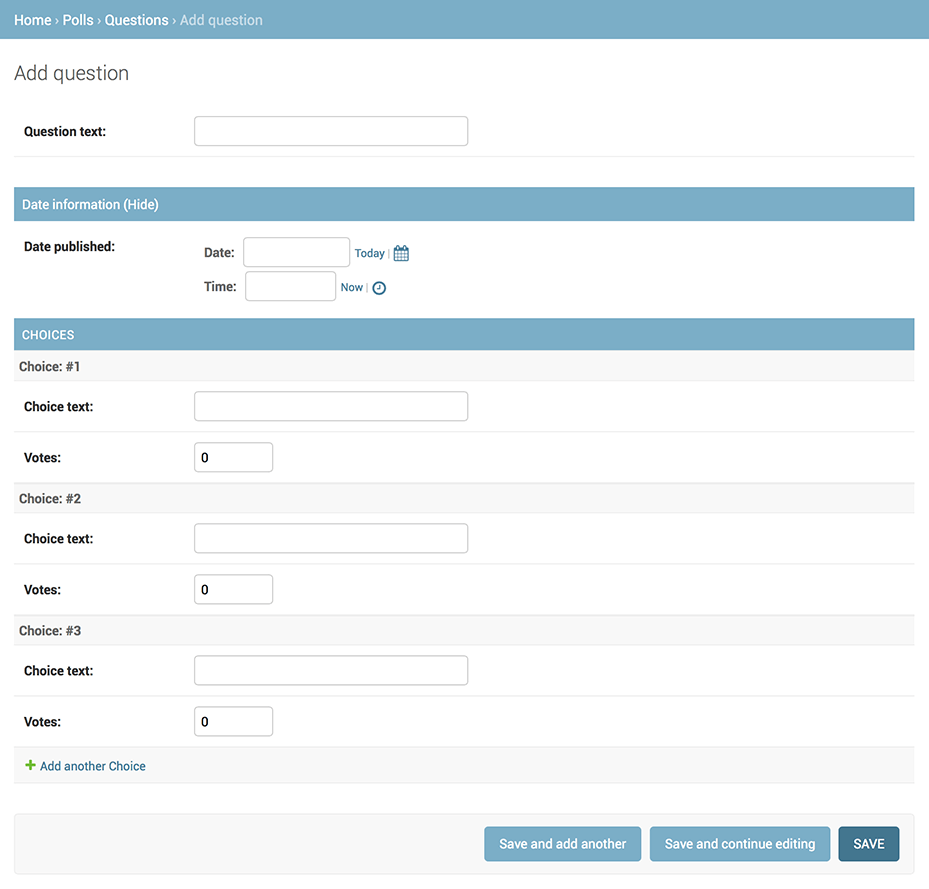
]

inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)

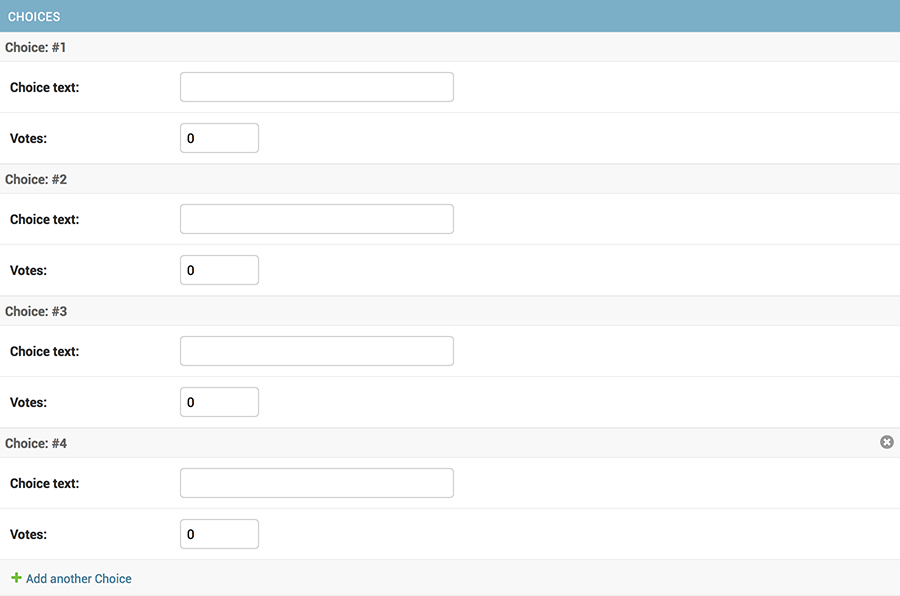
This tells Django: “Choice objects are edited on the Question admin page. By default, provide enough fields for 3 choices.”

Load the “Add question” 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.

At the end of the three current slots you will find an “Add another Choice” link. If you click on it, a new slot will be added. If you want to remove the added slot, you can click on the X to the top right of the added slot. Note that you can’t remove the original three slots. This image shows an added slot:



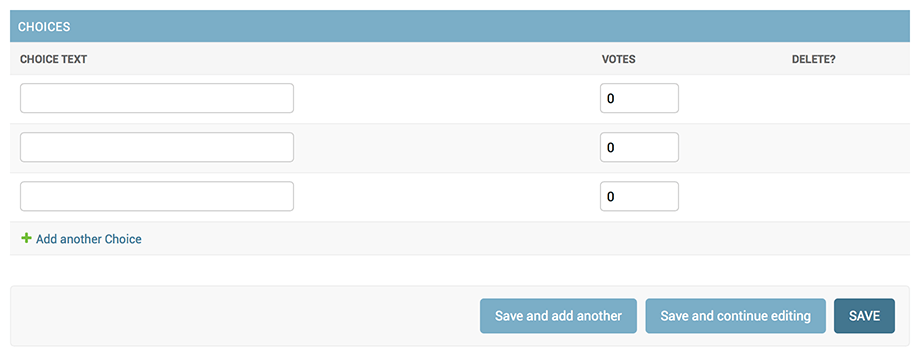
One small problem, though. It takes a lot of screen space to display all the fields for entering related Choice objects. For that reason, Django offers a tabular way of displaying inline related objects; you just need to change the ChoiceInline declaration to read:

polls/admin.py

class ChoiceInline(admin.TabularInline):

#...

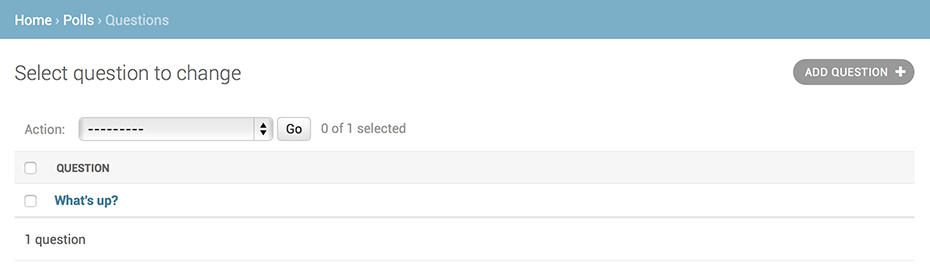
With that TabularInline (instead of StackedInline), the related objects are displayed in a more compact, table-based format:



Note that there is an extra “Delete?” column that allows removing rows added using the “Add Another Choice” button and rows that have already been saved.

## Customize the admin change list

Now that the Question admin page is looking good, let’s make some tweaks to the “change list” page – the one that displays all the questions in the system. Here’s what it looks like at this point:



By default, Django displays the str() of each object. Sometimes it’d be more helpful if we could display individual fields. Use [list\_display](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display) admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

polls/admin.py

class QuestionAdmin(admin.ModelAdmin):

# ...

list\_display = ('question\_text', 'pub\_date')

Also include was\_published\_recently() method from [Tutorial 2](https://docs.djangoproject.com/en/2.2/intro/tutorial02/):

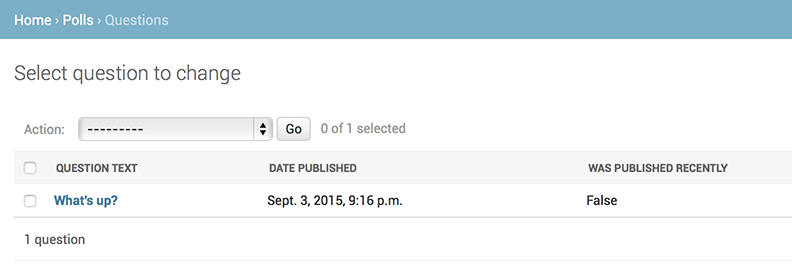
polls/admin.py

class QuestionAdmin(admin.ModelAdmin):

# ...

list\_display = ('question\_text', 'pub\_date', 'was\_published\_recently')

Now the question change list page looks like this:



You can click on the column headers to sort by those values – except in was\_published\_recently header, because sorting by the output of an arbitrary method is not supported. Note that the column header for was\_published\_recently is, by default, the name of the method (with underscores replaced with spaces), and that each line contains the string representation of the output. Improve by giving that method (in polls/models.py) a few attributes, as follows:

polls/models.py

class Question(models.Model):

# ...

def was\_published\_recently(self):

now = timezone.now()

return now - datetime.timedelta(days=1) <= self.pub\_date <= now

was\_published\_recently.admin\_order\_field = 'pub\_date'

was\_published\_recently.boolean = True

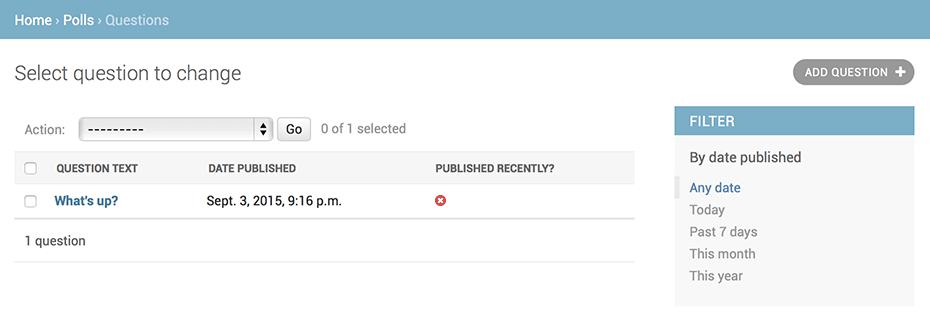
was\_published\_recently.short\_description = 'Published recently?'

For more information, see [list\_display](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display). Edit polls/admin.py and add an improvement to the Question change list page: filters using the [list\_filter](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_filter). Add the following line to:

QuestionAdmin:

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](https://docs.djangoproject.com/en/2.2/ref/models/fields/#django.db.models.DateTimeField), Django knows to give appropriate filter options: “Any date”, “Today”, “Past 7 days”, “This month”, “This year”. Add some search capability:

search\_fields = ['question\_text']

That adds a search box at the top of the change list. When somebody enters search terms, Django searches question\_text field. Use as many fields as you’d like – although because it uses a LIKE query behind the scenes, limiting the number of search fields to a reasonable number will make it easier for your database to do the search. Now’s also a good time to note that change lists give you free pagination. The default is to display 100 items per page. [Change list pagination](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_per_page), [search boxes](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.search_fields), [filters](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_filter), [date-hierarchies](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.date_hierarchy), and [column-header-ordering](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.ModelAdmin.list_display) all work together like you think they should.

## Customize the admin look and feel

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.

### Customizing your *project’s* templates

Create templates directory in your project directory (that contains manage.py). Templates can live anywhere filesystem that Django can access. (Django runs as whatever user your server runs.) But keep your templates within the project is good convention. Open your settings file (mysite/settings.py, remember) and add a [DIRS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES-DIRS) option in the [TEMPLATES](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES) setting:

mysite/settings.py

TEMPLATES = [

{

'BACKEND':

'django.template.backends.django.DjangoTemplates',

'DIRS': [os.path.join(BASE\_DIR, 'templates')],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors. messages',

],

},

},

]

[DIRS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES-DIRS) is a list of filesystem directories to check when loading Django templates; it’s a search path.

### Organizing templates

Just like the static files, we *could* have all templates together, in one big templates directory. But templates that belong to a particular application should be placed in that application’s template directory (e.g. polls/templates) rather than the project’s (templates). More detail in [reusable apps tutorial](https://docs.djangoproject.com/en/2.2/intro/reusable-apps/) *why*. Create directory admin inside templates. Copy template admin/base\_site.html from within default Django admin template directory in Django source code (django/contrib/admin/templates) into that directory.

### Where are the Django source files?

If you have difficulty finding where the Django source files are located on your system, run the following command:

$ python -c "import django; print(django.\_\_path\_\_)"

Then, edit file and replace {{ site\_header|default:\_('Django administration') }} (including the curly braces) with your site’s name as you see fit. You should end up with a section of code like:

{% block branding %}

<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>

{% endblock %}

This approach teaches you to override templates. In actual project, use [django.contrib.admin.AdminSite.site\_header](https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#django.contrib.admin.AdminSite.site_header) attribute to more easily make this particular customization. This template file contains lots of text like {% block branding %} and {{ title }}. The {% and {{ tags are part of Django’s template language. When Django renders admin/base\_site.html, this template language will be evaluated to produce the final HTML page, just like we saw in [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/). 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.

### Customizing your *application’s* templates

If [DIRS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES-DIRS) was empty by default, how did Django find the default admin templates? Answer: since [APP\_DIRS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES-APP_DIRS) is set to True, Django automatically looks for a templates/ subdirectory within each application package, for use as a fallback (don’t forget that django.contrib.admin is an application). It is more sensible to modify the *application’s* templates, rather than those in the *project*. Then you could include the polls application in any new project and be assured that it would find the custom templates it needed. See the [template loading documentation](https://docs.djangoproject.com/en/2.2/topics/templates/#template-loading) on how Django finds its templates.

### 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](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-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.

## What’s next?

The beginner tutorial ends here. In the meantime, you might want to check out some pointers on [where to go from here](https://docs.djangoproject.com/en/2.2/intro/whatsnext/). If you are familiar with Python packaging and interested in learning how to turn polls into a “reusable app”, check out [Advanced tutorial: How to write reusable apps](https://docs.djangoproject.com/en/2.2/intro/reusable-apps/).

# Advanced tutorial: How to write reusable apps

This advanced tutorial begins where [Tutorial 7](https://docs.djangoproject.com/en/2.2/intro/tutorial07/) left off. We’ll be turning our Web-poll into a standalone Python package you can reuse in new projects and share with other people. If you haven’t recently completed Tutorials 1–7, we encourage you to review these so that your example project matches the one described below.

## Reusability matters

It’s a lot of work to design, build, test and maintain a web application. Many Python and Django projects share common problems. Wouldn’t it be great if we could save some of this repeated work? Reusability is the way of life in Python. [The Python Package Index (PyPI)](https://pypi.org/) has a vast range of packages you can use in your own Python programs. Check out [Django Packages](https://djangopackages.org) for existing reusable apps you could incorporate in your project. Django itself is also just a Python package. This means that you can take existing Python packages or Django apps and compose them into your own web project. You only need to write the parts that make your project unique. Let’s say you were starting a new project that needed a polls app like the one we’ve been working on. How do you make this app reusable? Luckily, you’re well on the way already. In [Tutorial 1](https://docs.djangoproject.com/en/2.2/intro/tutorial01/), we saw how we could decouple polls from the project-level URLconf using an include. In this tutorial, we’ll take further steps to make the app easy to use in new projects and ready to publish for others to install and use.

### Package? App?

A Python [package](https://docs.python.org/3/glossary.html#term-package) provides a way of grouping related Python code for easy reuse. A package contains one or more files of Python code (also known as “modules”). A package can be imported with import foo.bar or from foo import bar. For a directory (like polls) to form a package, it must contain a special file \_\_init\_\_.py, even if this file is empty. A Django *application* is just a Python package that is specifically intended for use in a Django project. An application may use common Django conventions, such as having models, tests, urls, and views submodules. Later on we use the term *packaging* to describe the process of making a Python package easy for others to install. It can be a little confusing, we know.

## Your project and your reusable app

After the previous tutorials, our project should look like this:

mysite/

manage.py

mysite/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

polls/

\_\_init\_\_.py

admin.py

migrations/

\_\_init\_\_.py

0001\_initial.py

models.py

static/

polls/

images/

background.gif

style.css

templates/

polls/

detail.html

index.html

results.html

tests.py

urls.py

views.py

templates/

admin/

base\_site.html

You created mysite/templates in [Tutorial 7](https://docs.djangoproject.com/en/2.2/intro/tutorial07/), and polls/templates in [Tutorial 3](https://docs.djangoproject.com/en/2.2/intro/tutorial03/). Now perhaps it is clearer why we chose to have separate template directories for the project and application: everything that is part of the polls application is in polls. It makes the application self-contained and easier to drop into a new project. The polls directory could now be copied into a new Django project and immediately reused. It’s not quite ready to be published though. For that, we need to package the app to make it easy for others to install.

## Installing some prerequisites

The current state of Python packaging is a bit muddled with various tools. For this tutorial, we’re going to use [setuptools](https://pypi.org/project/setuptools/) to build our package. It’s the recommended packaging tool (merged with the distribute fork). We’ll also be using [pip](https://pypi.org/project/pip/) to install and uninstall it. You should install these two packages now. If you need help, you can refer to [how to install Django with pip](https://docs.djangoproject.com/en/2.2/topics/install/#installing-official-release). You can install setuptools the same way.

## Packaging your app

Python *packaging* refers to preparing your app in a specific format that can be easily installed and used. Django itself is packaged very much like this. For a small app like polls, this process isn’t too difficult.

1. First, create a parent directory for polls, outside of your Django project. Call this directory django-polls. Choosing a name for your app. When choosing a name for your package, check resources like PyPI to avoid naming conflicts with existing packages. It’s often useful to prepend django- to your module name when creating a package to distribute. This helps others looking for Django apps identify your app as Django specific. Application labels (that is, the final part of the dotted path to application packages) *must* be unique in [INSTALLED\_APPS](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-INSTALLED_APPS). Avoid using the same label as any of the Django [contrib packages](https://docs.djangoproject.com/en/2.2/ref/contrib/), for example auth, admin, or messages.
2. Move the polls directory into the django-polls directory.
3. Create a file django-polls/README.rst with the following contents:

django-polls/README.rst

=====

Polls

=====

Polls is a simple Django app to conduct Web-based polls. For each question, visitors can choose between a fixed number of answers.

Detailed documentation is in the "docs" directory.

Quick start

-----------

1. Add "polls" to your INSTALLED\_APPS setting like this::

INSTALLED\_APPS = [

...

'polls',

]

2. Include the polls URLconf in your project urls.py like this::

path('polls/', include('polls.urls')),

3. Run `python manage.py migrate` to create the polls models.

4. Start the development server and visit http://127.0.0.1:8000/admin/

to create a poll (you'll need the Admin app enabled).

5. Visit http://127.0.0.1:8000/polls/ to participate in the poll.

1. Create a django-polls/LICENSE file. Choosing a license is beyond the scope of this tutorial, but suffice it to say that code released publicly without a license is *useless*. Django and many Django-compatible apps are distributed under the BSD license; however, you’re free to pick your own license. Just be aware that your licensing choice will affect who is able to use your code.
2. Next we’ll create a setup.py file which provides details about how to build and install the app. A full explanation of this file is beyond the scope of this tutorial, but the [setuptools docs](https://setuptools.readthedocs.io/en/latest/) have a good explanation. Create a file django-polls/setup.py with the following contents:

django-polls/setup.py[¶](https://docs.djangoproject.com/en/2.2/intro/reusable-apps/#id2)

import os

from setuptools import find\_packages, setup

with open(os.path.join(os.path.dirname(\_\_file\_\_), 'README.rst')) as readme:

README = readme.read()

# allow setup.py to be run from any path

os.chdir(os.path.normpath(os.path.join(os.path.abspath(\_\_file\_\_), os.pardir)))

setup(

name='django-polls',

version='0.1',

packages=find\_packages(),

include\_package\_data=True,

license='BSD License', # example license

description='A simple Django app to conduct Web-based polls.',

long\_description=README,

url='https://www.example.com/',

author='Your Name',

author\_email='yourname@example.com',

classifiers=[

'Environment :: Web Environment',

'Framework :: Django',

# replace "X.Y" as appropriate

'Framework :: Django :: X.Y',

'Intended Audience :: Developers',

'License :: OSI Approved :: BSD License', # example license

'Operating System :: OS Independent',

'Programming Language :: Python',

'Programming Language :: Python :: 3.5',

'Programming Language :: Python :: 3.6',

'Topic :: Internet :: WWW/HTTP',

'Topic :: Internet :: WWW/HTTP :: Dynamic Content',

],

)

1. Only Python modules and packages are included in the package by default. To include additional files, we’ll need to create a MANIFEST.in file. The setuptools docs referred to in the previous step discuss this file in more details. To include the templates, the README.rst and our LICENSE file, create a file django-polls/MANIFEST.in with the following contents:

django-polls/MANIFEST.in[¶](https://docs.djangoproject.com/en/2.2/intro/reusable-apps/#id3)

include LICENSE

include README.rst

recursive-include polls/static \*

recursive-include polls/templates \*

1. It’s optional, but recommended, to include detailed documentation with your app. Create an empty directory django-polls/docs for future documentation. Add an additional line to django-polls/MANIFEST.in:
2. recursive-include docs \*

Note that the docs directory won’t be included in your package unless you add some files to it. Many Django apps also provide their documentation online through sites like [readthedocs.org](https://readthedocs.org).

1. Try building your package with python setup.py sdist (run from inside django-polls). This creates a directory called dist and builds your new package, django-polls-0.1.tar.gz.

For more information on packaging, see Python’s [Tutorial on Packaging and Distributing Projects](https://packaging.python.org/tutorials/packaging-projects/).

## Using your own package

Since we moved the polls directory out of the project, it’s no longer working. We’ll now fix this by installing our new django-polls package.

### Installing as a user library

The following steps install django-polls as a user library. Per-user installs have a lot of advantages over installing the package system-wide, such as being usable on systems where you don’t have administrator access as well as preventing the package from affecting system services and other users of the machine.

Note that per-user installations can still affect the behavior of system tools that run as that user, so virtualenv is a more robust solution (see below).

1. To install the package, use pip (you already [installed it](https://docs.djangoproject.com/en/2.2/intro/reusable-apps/#installing-reusable-apps-prerequisites), right?):
2. pip install --user django-polls/dist/django-polls-0.1.tar.gz
3. With luck, your Django project should now work correctly again. Run the server again to confirm this.
4. To uninstall the package, use pip:
5. pip uninstall django-polls

## Publishing your app

Now that we’ve packaged and tested django-polls, it’s ready to share with the world! If this wasn’t just an example, you could now:

* Email the package to a friend.
* Upload the package on your website.
* Post the package on a public repository, such as [the Python Package Index (PyPI)](https://pypi.org/). [packaging.python.org](https://packaging.python.org) has [a good tutorial](https://packaging.python.org/tutorials/packaging-projects/#uploading-the-distribution-archives) for doing this.

## Installing Python packages with virtualenv

Earlier, we installed the polls app as a user library. This has some disadvantages:

* Modifying the user libraries can affect other Python software on your system.
* You won’t be able to run multiple versions of this package (or others with the same name).

Typically, these situations only arise once you’re maintaining several Django projects. When they do, the best solution is to use [virtualenv](https://virtualenv.pypa.io/). This tool allows you to maintain multiple isolated Python environments, each with its own copy of the libraries and package namespace.

## What to read next

So you’ve read all the [introductory material](https://docs.djangoproject.com/en/2.2/intro/) and have decided you’d like to keep using Django. We’ve only just scratched the surface with this intro (in fact, if you’ve read every single word, you’ve read about 5% of the overall documentation). So what’s next? Well, we’ve always been big fans of learning by doing. At this point you should know enough to start a project of your own and start fooling around. As you need to learn new tricks, come back to the documentation. We’ve put a lot of effort into making Django’s documentation useful, easy to read and as complete as possible. The rest of this document explains more about how the documentation works so that you can get the most out of it. (Yes, this is documentation about documentation. Rest assured we have no plans to write a document about how to read the document about documentation.)

### Finding documentation

Django’s got a *lot* of documentation – almost 450,000 words and counting – so finding what you need can sometimes be tricky. A few good places to start are the [Search Page](https://docs.djangoproject.com/en/2.2/search/) and the [Index](https://docs.djangoproject.com/en/2.2/genindex/).

Or you can just browse around!

### How the documentation is organized

Django’s main documentation is broken up into “chunks” designed to fill different needs:

* The [introductory material](https://docs.djangoproject.com/en/2.2/intro/) is designed for people new to Django – or to Web development in general. It doesn’t cover anything in depth, but instead gives a high-level overview of how developing in Django “feels”.
* The [topic guides](https://docs.djangoproject.com/en/2.2/topics/), on the other hand, dive deep into individual parts of Django. There are complete guides to Django’s [model system](https://docs.djangoproject.com/en/2.2/topics/db/), [template engine](https://docs.djangoproject.com/en/2.2/topics/templates/), [forms framework](https://docs.djangoproject.com/en/2.2/topics/forms/), and much more. This is probably where you’ll want to spend most of your time; if you work your way through these guides you should come out knowing pretty much everything there is to know about Django.
* Web development is often broad, not deep – problems span many domains. We’ve written a set of [how-to guides](https://docs.djangoproject.com/en/2.2/howto/) that answer common “How do I …?” questions. Here you’ll find information about [generating PDFs with Django](https://docs.djangoproject.com/en/2.2/howto/outputting-pdf/), [writing custom template tags](https://docs.djangoproject.com/en/2.2/howto/custom-template-tags/), and more. Answers to really common questions can also be found in the [FAQ](https://docs.djangoproject.com/en/2.2/faq/).
* The guides and how-to’s don’t cover every single class, function, and method available in Django – that would be overwhelming when you’re trying to learn. Instead, details about individual classes, functions, methods, and modules are kept in the [reference](https://docs.djangoproject.com/en/2.2/ref/). This is where you’ll turn to find the details of a particular function or whatever you need.
* If you are interested in deploying a project for public use, our docs have [several guides](https://docs.djangoproject.com/en/2.2/howto/deployment/) for various deployment setups as well as a [deployment checklist](https://docs.djangoproject.com/en/2.2/howto/deployment/checklist/) for some things you’ll need to think about.
* Finally, there’s some “specialized” documentation not usually relevant to most developers. This includes the [release notes](https://docs.djangoproject.com/en/2.2/releases/) and [internals documentation](https://docs.djangoproject.com/en/2.2/internals/) for those who want to add code to Django itself, and a [few other things that simply don’t fit elsewhere](https://docs.djangoproject.com/en/2.2/misc/).

### How documentation is updated

Just as the Django code base is developed and improved on a daily basis, our documentation is consistently improving. We improve documentation for several reasons:

* To make content fixes, such as grammar/typo corrections.
* To add information and/or examples to existing sections that need to be expanded.
* To document Django features that aren’t yet documented. (The list of such features is shrinking but exists nonetheless.)
* To add documentation for new features as new features get added, or as Django APIs or behaviors change.

Django’s documentation is kept in the same source control system as its code. It lives in the [docs](https://github.com/django/django/blob/stable/2.2.x/docs) directory of our Git repository. Each document online is a separate text file in the repository.

### Where to get it

You can read Django documentation in several ways. They are, in order of preference:

#### On the Web

The most recent version of the Django documentation lives at <https://docs.djangoproject.com/en/dev/>. These HTML pages are generated automatically from the text files in source control. That means they reflect the “latest and greatest” in Django – they include the very latest corrections and additions, and they discuss the latest Django features, which may only be available to users of the Django development version. (See [Differences between versions](https://docs.djangoproject.com/en/2.2/intro/whatsnext/#differences-between-doc-versions) below.)

We encourage you to help improve the docs by submitting changes, corrections and suggestions in the [ticket system](https://code.djangoproject.com/). The Django developers actively monitor the ticket system and use your feedback to improve the documentation for everybody.

Note, however, that tickets should explicitly relate to the documentation, rather than asking broad tech-support questions. If you need help with your particular Django setup, try the [django-users](https://docs.djangoproject.com/en/2.2/internals/mailing-lists/#django-users-mailing-list) mailing list or the [#django IRC channel](irc://irc.freenode.net/django) instead.

#### In plain text

For offline reading, or just for convenience, you can read the Django documentation in plain text.

If you’re using an official release of Django, the zipped package (tarball) of the code includes a docs/ directory, which contains all the documentation for that release.

If you’re using the development version of Django (aka the master branch), the docs/ directory contains all of the documentation. You can update your Git checkout to get the latest changes.

One low-tech way of taking advantage of the text documentation is by using the Unix grep utility to search for a phrase in all of the documentation. For example, this will show you each mention of the phrase “max\_length” in any Django document:

$ grep -r max\_length /path/to/django/docs/

#### As HTML, locally

You can get a local copy of the HTML documentation following a few easy steps:

* Django’s documentation uses a system called [Sphinx](http://sphinx-doc.org/) to convert from plain text to HTML. You’ll need to install Sphinx by either downloading and installing the package from the Sphinx website, or with pip:

$ pip install Sphinx

* Then, just use the included Makefile to turn the documentation into HTML:

$ cd path/to/django/docs

$ make html

You’ll need [GNU Make](https://www.gnu.org/software/make/) installed for this.

If you’re on Windows you can alternatively use the included batch file:

cd path\to\django\docs

make.bat html

* The HTML documentation will be placed in docs/\_build/html.

### Differences between versions

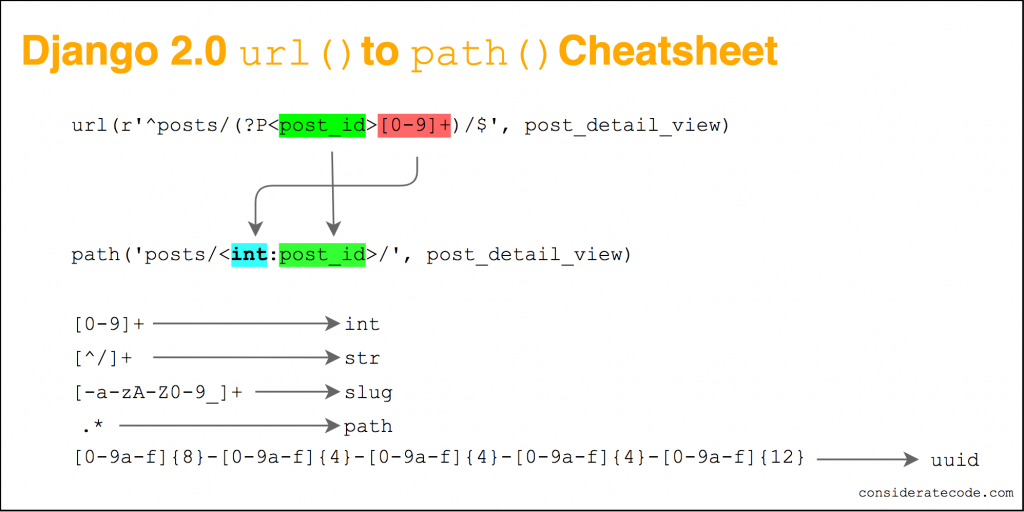
The text documentation in the master branch of the Git repository contains the “latest and greatest” changes and additions. These changes include documentation of new features targeted for Django’s next [feature release](https://docs.djangoproject.com/en/2.2/internals/release-process/#term-feature-release). For that reason, it’s worth pointing out our policy to highlight recent changes and additions to Django.

We follow this policy:

* The development documentation at <https://docs.djangoproject.com/en/dev/> is from the master branch. These docs correspond to the latest feature release, plus whatever features have been added/changed in the framework since then.
* As we add features to Django’s development version, we update the documentation in the same Git commit transaction.
* To distinguish feature changes/additions in the docs, we use the phrase: “New in Django Development version” for the version of Django that hasn’t been released yet, or “New in version X.Y” for released versions.
* Documentation fixes and improvements may be backported to the last release branch, at the discretion of the committer, however, once a version of Django is [no longer supported](https://docs.djangoproject.com/en/2.2/internals/release-process/#supported-versions-policy), that version of the docs won’t get any further updates.
* The [main documentation Web page](https://docs.djangoproject.com/en/dev/) includes links to documentation for previous versions. Be sure you are using the version of the docs corresponding to the version of Django you are using!

# Django 2.0 url() to path() cheatsheet

[May 2, 2018](https://consideratecode.com/2018/05/02/django-2-0-url-to-path-cheatsheet/)[Daniel Hepper](https://consideratecode.com/author/daniel/)[Django](https://consideratecode.com/category/django/), [Web Development](https://consideratecode.com/category/web-development/)



Django 2.0 introduced a new way to define URLs, which greatly simplifies how parameters are captured.

In earlier versions of Django, you had to use the [url()](https://docs.djangoproject.com/en/1.11/ref/urls/#url) method and pass a regular expressions with named capturing groups to capture URL parameters.

url(r'^posts/(?P<post\_id>[0-9]+)/$', post\_detail\_view)

In Django 2.0, you use the [path()](https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.path) method with path converters to capture URL parameters.

path('posts/<int:post\_id>/', post\_detail\_view)

**path() always matches the complete path**, so path('account/login/') is equivalent to url('^account/login/$').

The part in angle brackets (<int:post\_id>) captures a URL parameter that is passed to a view. The part after the colon (post\_id) defines the name of the parameter, the part before the colon (int) names a path converter.

Path converters are not only easier to read, they also bring a new feature: **path converters can convert parameters to the appropriate type** before passing it to the view.

Earlier versions of Django just passed the matched string to the view, meaning you had to write this:

url(r'^posts/(?P<post\_id>[0-9]+)/$', post\_detail\_view)

def post\_detail\_view(request, post\_id):

post\_id = int(post\_id)

...

Starting with Django 2.0, you would just write this:

path('posts/<int:post\_id>')

def post\_detail\_view(request, post\_id):

...

Django 2.0 comes with five built-in [converters](https://github.com/django/django/blob/stable/2.0.x/django/urls/converters.py):

* **str**  
  Matches any non-empty string, excluding the path separator, ‘/’. This is the default if a converter isn’t included in the expression.  
  Example: 'best\_product\_ever!-123.html'  
  Returns a string (str).  
  Equivalent regular expression: [^/]+
* **int**  
  Matches zero or any positive integer.  
  Example: '1234'  
  Returns an integer (int).  
  Equivalent regular expression: [0-9]+
* **slug**  
  Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters.  
  Returns a string (str).  
  Example: 'building-your-1st-django-site'  
  Equivalent regular expression: [-a-zA-Z0-9\_]+
* **uuid**  
  Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase.  
  Returns a UUID instance ([uuid.UUID](https://docs.python.org/3/library/uuid.html#uuid.UUID)).  
  Example: '075194d3-6885-417e-a8a8-6c931e272f00'  
  Equivalent regular expression: [0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}
* **path**  
  Matches any non-empty string, including the path separator, ‘/’. This allows you to match against a complete URL path rather than just a segment of a URL path as with str.  
  Example: '/path/to/file'  
  Returns a string (str).  
  Equivalent regular expression: '.+'

If the pre-defined path converters don’t suit your needs, you can [register custom path converters](https://docs.djangoproject.com/en/2.0/topics/http/urls/#registering-custom-path-converters), which can do tricks like [converting a URL parameter straight to a model instance](https://consideratecode.com/2018/05/11/the-hidden-powers-of-custom-django-2-0-path-converters/). Alternatively, you can still use good ol’ regular expression by using [re\_path()](https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.re_path) instead of [path()](https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.path).

from django.urls import re\_path

re\_path(r'posts/(?P<post\_id>[0-9]+)/$', post\_detail\_view)

By the way, the old url() function is still available, so there is no need to change all your URL definitions right now.

I think the new path() method greatly simplifies dealing with URLs. I’ve been working with Django for more than 10 years now and still can’t remember how to define a named regular expression capturing group. Furthermore, manually doing the type conversion violates the DRY-principle.

I hope this little cheat sheet helps you during the transition to Django 2.0. Be sure to check out my follow-up post on [how to create powerful custom path converters](https://consideratecode.com/2018/05/11/the-hidden-powers-of-custom-django-2-0-path-converters/). Do you have any further question? Don’t hesitate to leave a comment below.

# DJANGO REST Framework

## [Example](https://www.django-rest-framework.org/#example)

Let's take a look at a quick example of using REST framework to build a simple model-backed API. We'll create a read-write API for accessing information on the users of our project. Any global settings for a REST framework API are kept in a single configuration dictionary named REST\_FRAMEWORK. Start off by adding the following to your settings.py module:

REST\_FRAMEWORK = {

# Use Django's standard `django.contrib.auth` permissions,

# or allow read-only access for unauthenticated users.

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions. DjangoModelPermissionsOrAnonReadOnly'

]

}

Don't forget to make sure you've also added rest\_framework to your INSTALLED\_APPS. We're ready to create our API now. Here's our project's root urls.py module:

from django.conf.urls import url, include

from django.contrib.auth.models import User

from rest\_framework import routers, serializers, viewsets

# Serializers define the API representation.

class UserSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = User

fields = ('url', 'username', 'email', 'is\_staff')

# ViewSets define the view behavior.

class UserViewSet(viewsets.ModelViewSet):

queryset = User.objects.all()

serializer\_class = UserSerializer

# Routers provide an easy way of automatically determining the URL conf.

router = routers.DefaultRouter()

router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.

# Additionally, we include login URLs for the browsable API.

urlpatterns = [

url(r'^', include(router.urls)),

url(r'^api-auth/', include('rest\_framework.urls', namespace='rest\_framework'))

]

Now open the API in your browser at <http://127.0.0.1:8000/>, and view your new 'users' API. If you use the login control in the top right corner, you can add, create and delete system users.

## [Quickstart](https://www.django-rest-framework.org/tutorial/quickstart/#quickstart)

We're going to create a simple API to allow admin users to view and edit the users and groups in the system.

### [Project setup](https://www.django-rest-framework.org/tutorial/quickstart/#project-setup)

Create a new Django project named tutorial, then start a new app called quickstart.

# Create the project directory

mkdir tutorial

cd tutorial

# Create a virtualenv to isolate our package dependencies locally

virtualenv env

source env/bin/activate # On Windows use `env\Scripts\activate`

# Install Django and Django REST framework into the virtualenv

pip install django

pip install djangorestframework

# Set up a new project with a single application

django-admin startproject tutorial . # Note the trailing '.' character

cd tutorial

django-admin startapp quickstart

cd ..

The project layout should look like:

$ pwd

<some path>/tutorial

$ find .

.

./manage.py

./tutorial

./tutorial/\_\_init\_\_.py

./tutorial/quickstart

./tutorial/quickstart/\_\_init\_\_.py

./tutorial/quickstart/admin.py

./tutorial/quickstart/apps.py

./tutorial/quickstart/migrations

./tutorial/quickstart/migrations/\_\_init\_\_.py

./tutorial/quickstart/models.py

./tutorial/quickstart/tests.py

./tutorial/quickstart/views.py

./tutorial/settings.py

./tutorial/urls.py

./tutorial/wsgi.py

It may look unusual that the application has been created within the project directory. Using the project's namespace avoids name clashes with external modules (a topic that goes outside the scope of the quickstart). Now sync your database for the first time:

python manage.py migrate

We'll also create an initial user named admin with a password of password123. We'll authenticate as that user later in our example.

python manage.py createsuperuser --email admin@example.com --username admin

Once you've set up a database and the initial user is created and ready to go, open up the app's directory and we'll get coding...

### [Serializers](https://www.django-rest-framework.org/tutorial/quickstart/#serializers)

First up we're going to define some serializers. Create a module named tutorial/quickstart/serializers.py that we'll use for our data representations.

from django.contrib.auth.models import User, Group

from rest\_framework import serializers

class UserSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = User

fields = ('url', 'username', 'email', 'groups')

class GroupSerializer(serializers.HyperlinkedModelSerializer):

class Meta:

model = Group

fields = ('url', 'name')

Notice we use hyperlinked relationsHyperlinkedModelSerializer. You can also use primary key and various other relationships, but hyperlinking is good RESTful design.

### [Views](https://www.django-rest-framework.org/tutorial/quickstart/#views)

Open tutorial/quickstart/views.py and get typing.

from django.contrib.auth.models import User, Group

from rest\_framework import viewsets

from tutorial.quickstart.serializers import UserSerializer, GroupSerializer

class UserViewSet(viewsets.ModelViewSet):

"""

API endpoint that allows users to be viewed or edited.

"""

queryset = User.objects.all().order\_by('-date\_joined')

serializer\_class = UserSerializer

class GroupViewSet(viewsets.ModelViewSet):

"""

API endpoint that allows groups to be viewed or edited.

"""

queryset = Group.objects.all()

serializer\_class = GroupSerializer

Rather than write multiple views, group all the common behavior into classes called ViewSets. We can easily break these down into individual views if we need to, but using viewsets keeps the view logic nicely organized as well as being very concise.

### [URLs](https://www.django-rest-framework.org/tutorial/quickstart/#urls)

Okay, now let's wire up the API URLs. On to tutorial/urls.py...

from django.urls import include, path

from rest\_framework import routers

from tutorial.quickstart import views

router = routers.DefaultRouter()

router.register(r'users', views.UserViewSet)

router.register(r'groups', views.GroupViewSet)

# Wire up our API using automatic URL routing.

# Additionally, we include login URLs for the browsable API.

urlpatterns = [

path('', include(router.urls)),

path('api-auth/', include('rest\_framework.urls', namespace='rest\_framework'))

]

Because we're using viewsets instead of views, we can automatically generate the URL conf for our API, by simply registering the viewsets with a router class. Again, if we need more control over the API URLs we can simply drop down to using regular class-based views, and writing the URL conf explicitly. Finally, we're including default login and logout views for use with the browsable API. That's optional, but useful if your API requires authentication and you want to use the browsable API.

### [Pagination](https://www.django-rest-framework.org/tutorial/quickstart/#pagination)

Pagination allows you to control how many objects per page are returned. To enable it add the following lines to tutorial/settings.py

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10

}

### [Settings](https://www.django-rest-framework.org/tutorial/quickstart/#settings)

Add 'rest\_framework' to INSTALLED\_APPS. The settings module will be in tutorial/settings.py

INSTALLED\_APPS = (

...

'rest\_framework',

)

Okay, we're done.

### [Testing our API](https://www.django-rest-framework.org/tutorial/quickstart/#testing-our-api)

We're now ready to test the API we've built. Let's fire up the server from the command line.

python manage.py runserver

We can now access our API, both from the command-line, using tools like curl...

bash: curl -H 'Accept: application/json; indent=4' -u admin:password123 http://127.0.0.1:8000/users/

{

"count": 2,

"next": null,

"previous": null,

"results": [

{

"email": "admin@example.com",

"groups": [],

"url": "http://127.0.0.1:8000/users/1/",

"username": "admin"

},

{

"email": "tom@example.com",

"groups": [ ],

"url": "http://127.0.0.1:8000/users/2/",

"username": "tom"

}

]

}

Or using the [httpie](https://github.com/jakubroztocil/httpie#installation), command line tool...

bash: http -a admin:password123 http://127.0.0.1:8000/users/

HTTP/1.1 200 OK

...

{

"count": 2,

"next": null,

"previous": null,

"results": [

{

"email": "admin@example.com",

"groups": [],

"url": "http://localhost:8000/users/1/",

"username": "paul"

},

{

"email": "tom@example.com",

"groups": [ ],

"url": "http://127.0.0.1:8000/users/2/",

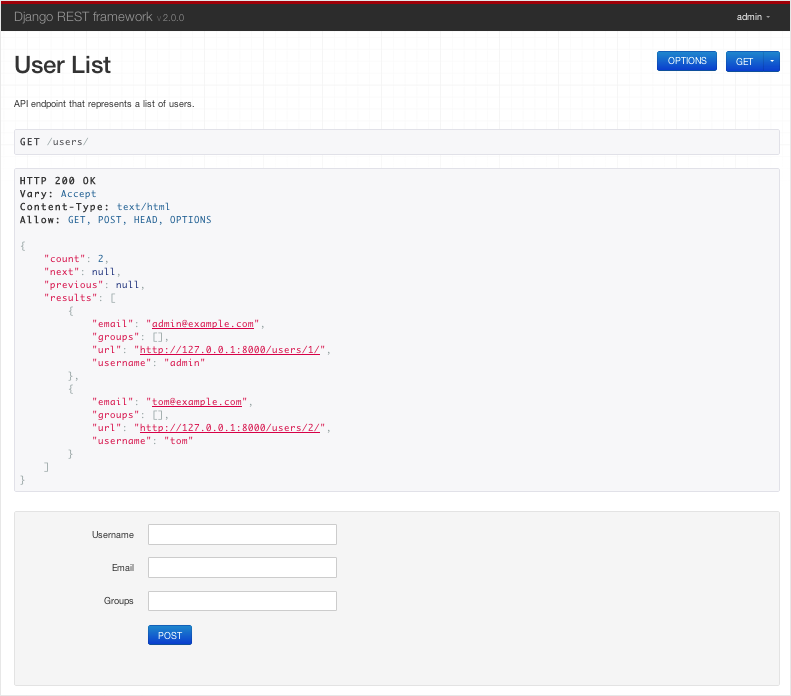
"username": "tom"

}

]

}

Or through rowser, to URLhttp://127.0.0.1:8000/users/



If you're working through the browser, make sure to login using the control in the top right corner. Great, that was easy! To get a more in depth understanding of how REST framework fits together head on over to [the tutorial](https://www.django-rest-framework.org/tutorial/1-serialization/), or start browsing the [API guide](https://www.django-rest-framework.org/#api-guide).

# API development using Django REST

[Django REST framework](https://micropyramid.com/django-rest-framework-development-services/) is  a best toolkit to create an API. It supports both ORM and Non-ORM data sources. It can support regular function based view and class based views.

1) Installation of Django REST framework.

pip install djangorestframework

2) Add 'rest\_framework'  in 'INSTALLED\_APPS'

#### settings.py

INSTALLED\_APPS = (

...

'rest\_framework',

...

)

3) Configure Django REST framework with: '[REST\_FRAMEWORK'](https://github.com/tomchristie/django-rest-framework/blob/master/rest_framework/settings.py#L29)

Default configurations can be overriden:

#### settings.py

REST\_FRAMEWORK = {

....

'DEFAULT\_AUTHENTICATION\_CLASSES': (

'rest\_framework.authentication.BasicAuthentication',

'rest\_framework.authentication.SessionAuthentication',

),

....

}

The *Django REST framework*  is very similar to *Django.* Create your app and add it to INSTALLED\_APPS in settings.py. Below we are writing both function and class based views..

#### models.py

**from** **django.contrib.auth.models** **import** AbstractBaseUser, PermissionsMixin

GENDER\_CHOICES = (

('Male', 'Male'),

('Female', 'Female'),

('Other', 'Other')

)

**class** User(AbstractBaseUser, PermissionsMixin):

first\_name = models.CharField(max\_length=100)

last\_name = models.CharField(max\_length=100)

email = models.EmailField(unique=True)

username = models.CharField(max\_length=100, blank=True, null=True)

is\_active = models.BooleanField(default=True)

is\_staff = models.BooleanField(default=False)

dob = models.DateField(null=True)

phone = models.CharField(max\_length=20, null=True)

gender = models.CharField(choices=GENDER\_CHOICES, max\_length=6)

address = models.TextField()

password = models.CharField(maxlength=255)

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

**return** self.email

#### urls.py

**from** **django.conf.urls** **import** url

**from** **.** **import** views

urlpatterns = [

url(r'^user/create/$', views.UserCreateView.as\_view(), name="user\_create"),

url(r'^users/list/$', views.UsersListView.as\_view(), name="users\_list"),

url(r'^users/(?P<pk>\d+)/detail/$', views.UserDetailView.as\_view(), name="user\_detail"),

url(r'^users/(?P<pk>\d+)/update/$', views.UserUpdateView.as\_view(), name="user\_update"),

url(r'^users/(?P<pk>\d+)/delete/$', views.UserDeleteView.as\_view(), name="user\_delete"),

]

## 1. User Create View

serializer for validation( similar to django forms)

#### serializers.py

**from** **rest\_framework** **import** serializers

**from** **.models** **import** GENDER\_CHOICES, User

*# normal serializer [similar to forms.Form]*

**class** UserSerializer(serializers.Serializer):

first\_name = serializers.CharField(max\_length=100)

last\_name = serializers.CharField(max\_length=100)

email = serializers.EmailField()

username = serializers.CharField(max\_length=100)

dob = serializers.DateField()

phone = serializers.CharField(max\_length=20)

gender = serializers.ChoiceField(choices=GENDER\_CHOICES)

address = serializers.TextField()

password = serializers.CharField(max\_length=255)

*# Called if we save serializer if it do not have an instance*

**def** create(self, validated\_data):

password = validated\_data.pop("password")

user = User.objects.create(\*\*validated\_data)

**if** password:

user.set\_password(password)

user.save()

**return** user

*# is called if we save serializer if it have an instance*

**def** update(self, instance, validated\_data):

password = validated\_data.pop("password")

instance.\_\_dict\_\_.update(validated\_data)

**if** password:

instance.set\_password(password)

instance.save()

**return** instance

*# model serializer [similar to forms.ModelForm]*

**class** UserSerializer(serializers.ModelSerializer):

**class** Meta:

model = User

fields = ("first\_name", "last\_name", "email", "username", "dob", "phone", "gender", "address")

#### generic\_views.py

**from** **rest\_framework** **import** generics

**from** **.serilizers** **import** UserSerializer

**class** UserCreateView(generics.CreateAPIView):

serializer\_class = UserSerializer

#### function\_views.py

**from** **rest\_framework.decorators** **import** api\_view

**from** **rest\_framework.response** **import** Response

**from** **.serilizers** **import** UserSerializer

@api\_view(["POST"])

**def** create\_user(request):

serializer = UserSerializer(request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response({"message": "User created"})

**else**:

data = {

"error": True,

"errors": serializer.errors,

}

**return** Response(data)

## 2. User Detail View

#### generic\_views.py

**from** **rest\_framework** **import** generics

**from** **.serilizers** **import** UserSerializer

**class** UserDetailView(generics.RetrieveAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

#### function\_views.py

**from** **rest\_framework.decorators** **import** api\_view

**from** **rest\_framework.response** **import** Response

**from** **.serilizers** **import** UserSerializer

**from** **.models** **import** \*

@api\_view(["GET"])

**def** user\_details(request, pk):

user = User.objects.get(id=pk)

serializer = UserSerializer(user)

**return** Response(serializer.data)

## 3. User Update View

#### generic\_views.py

**from** **rest\_framework** **import** generics

**from** **.serilizers** **import** UserSerializer

**class** UserUpdateView(generics.RetrieveUpdateAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

#### function\_views.py

**from** **rest\_framework.decorators** **import** api\_view

**from** **rest\_framework.response** **import** Response

**from** **.serilizers** **import** UserSerializer

**from** **.models** **import** \*

@api\_view(["GET", "PUT"])

**def** user\_update(request, pk):

user = User.objects.get(id=pk)

**if** request.method == "PUT":

serializer = UserSerializer(user, data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data)

**else**:

**return** Response({"error": serializer.errors, "error": True})

serializer = UserSerializer(user)

**return** Response(serializer.data)

## 4. User List View

#### generic\_views.py

**from** **rest\_framework** **import** generics

**from** **.serilizers** **import** UserSerializer

**class** UsersListView(generics.ListAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

#### function\_views.py

**from** **rest\_framework.decorators** **import** api\_view

**from** **rest\_framework.response** **import** Response

**from** **.serilizers** **import** UserSerializer

**from** **.models** **import** \*

@api\_view(["GET"])

**def** users\_list(request):

users = User.objects.all()

serializer = UserSerializer(users, many=True)

**return** Response(serializer.data)

## 5. User DeleteView

#### generic\_views.py

**from** **rest\_framework** **import** mixins

from.models **import** \*

**class** UserDeleteView(mixins.RetrieveModelMixin, mixins.DestroyModelMixin):

queryset = User.objects.all()

**def** get(self, request, \*args, \*\*kwargs):

**return** self.destroy(request, \*args, \*\*kwargs)

#### function\_views.py

**from** **.models** **import** \*

**from** **rest\_framework.decorators** **import** api\_view

**from** **rest\_framework.response** **import** Response

**def** delete\_user(request, pk):

user = get\_object\_or\_404(User, id=pk)

user.delete()

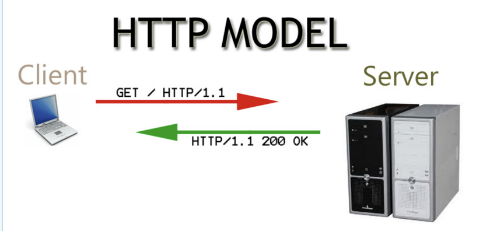
**return** Response({"message": "Deleted"})

More implementation details: [***http://www.django-rest-framework.org/#tutorial***](http://www.django-rest-framework.org/#tutorial) . More about our Django CRM (Customer Relationship Management) Open Source Package. [Check Code](https://github.com/MicroPyramid/Django-CRM)

# Build API with Django REST and Class-Based Views

[Kyle Truong](https://codeburst.io/@ktruong008) Jun 8, 2017

With the rise of Single-Page-Applications and the trend of separating monoliths into services with distinct front and backends, knowing how to make your own RESTful API for your backend is more important than ever. Officially, a **RESTful API** = **Representational State Transfer-ful Application Programming Interface**. Big words, but what it really boils down to is putting data onto your web server in a way that’s accessible to other servers and clients, and it works through HTTP requests and responses and carefully structured URL routes to represent specific resource(s). It looks a lot like this:



<http://wiki.hashphp.org/HttpPrimer>

**HTTP** is short for **Hypertext Transfer Protocol** and it’s a set of rules that dictate how data is packaged and communicated throughout the web. There are other protocols that go along with HTTP, but HTTP will be the focus here, for simplicity. An HTTP request looks something like this:

POST /cgi-bin/process.cgi HTTP/1.1  
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)  
Host: [www.tutorialspoint.com](http://www.tutorialspoint.com/)  
Content-Type: application/x-www-form-urlencoded  
Content-Length: **length**  
Accept-Language: en-us  
Accept-Encoding: gzip, deflate  
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string

[*https://www.tutorialspoint.com/http/http\_requests.htm*](https://www.tutorialspoint.com/http/http_requests.htm)

And an HTTP response looks something like this:

HTTP/1.1 200 OK  
Date: Mon, 27 Jul 2009 12:28:53 GMT  
Server: Apache/2.2.14 (Win32)  
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT  
Content-Length: 88  
Content-Type: text/html  
Connection: Closed

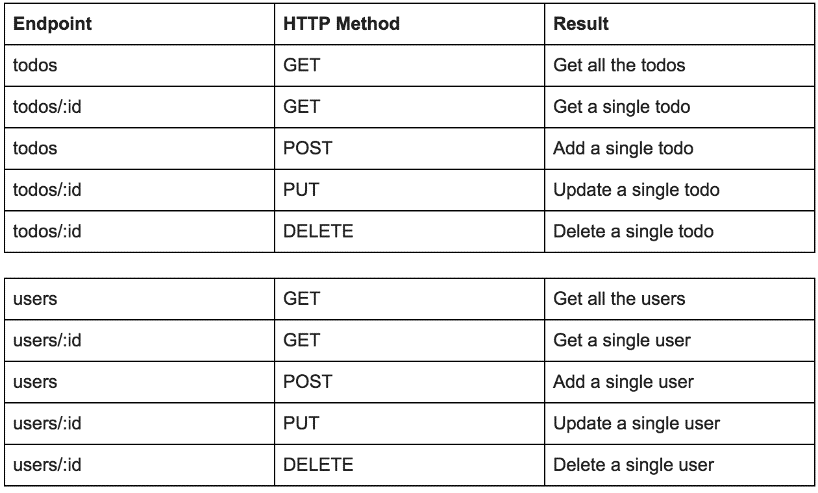
<html>  
<body>  
<h1>Hello, World!</h1>  
</body>  
</html>

<https://www.tutorialspoint.com/http/http_responses.htm>

Clients send HTTP requests to the server and the server sends back HTTP responses. Many things can fit within requests and responses, though most of the times the data is either metadata about the request/response or some kind of JSON string or both, when dealing with APIs. This is not a comprehensive guide, so you should be somewhat familiar with making AJAX calls with JavaScript and Django itself. You can make an API in any modern language and the concepts behind it are similar but I am choosing Python because:

* Django and Django Rest Framework (DRF) are both mature, stable, and well-documented
* Django and DRF gives you a lot out of the box (pluggable auth systems, serializers, ORMs, Views…)
* Highly customizable on every level (DRF)

In a nutshell, this is what we’ll be creating:



**Table 1**

We structure our endpoints in accordance with common RESTful guidelines to have clear endpoints that return expected resources.

## Setup

Use *virtualenv* and *virtualenvwrapper*. Install Django, Django Rest Framework, and coreapi (for Django Rest Framework)

ktruong$ which python3

/usr/local/bin/python3

ktruong$ mkvirtualenv — python=/usr/local/bin/python3 auth-api

…

ktruong$ pip install django djangorestframework coreapi

Now, create django project, make folders, and adjust project settings:

(ktruong$ pwd

/Users/ktruong/git\_projects/blogs/auth-api

ktruong$ touch README.md

ktruong$ django-admin.py startproject auth\_api

ktruong$ cd auth\_api

ktruong$ ls

auth\_api manage.py

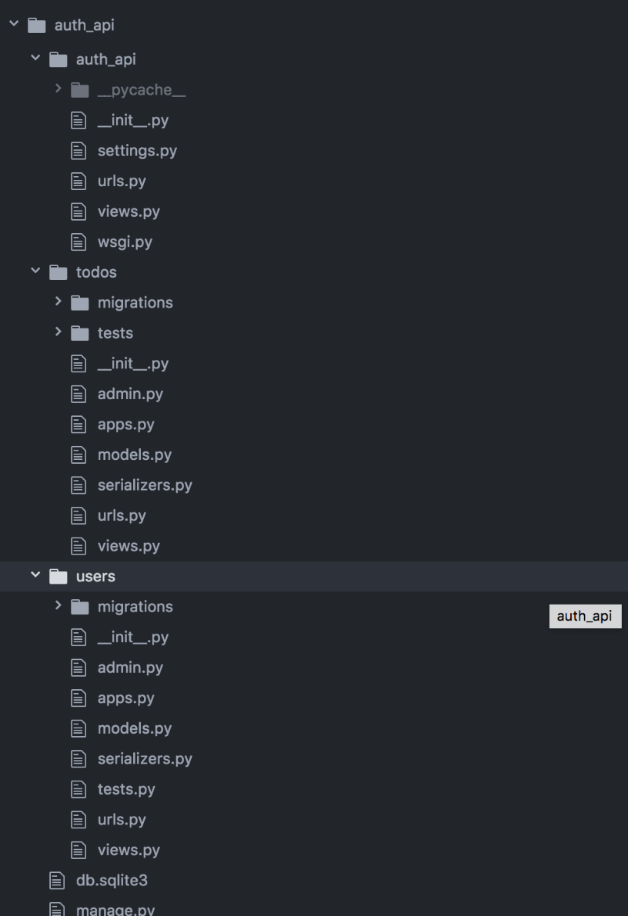
ktruong$ python manage.py startapp users

ktruong$ python manage.py startapp todos

ktruong$ ls

auth\_api manage.py todos users

ktruong$



### settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework',

'todos',

'users',

]

AUTH\_USER\_MODEL = 'users.User'

Notice how we explicitly set our *AUTH\_USER\_MODEL*to a custom *User* model (we’ll write the actual model, *users.User*, later). We could use the default *User*model that comes from Django but it becomes unnecessarily complicated to change it down the road. A solution is to write our own *User* model that subclasses the same *AbstractUser* model that Django’s *User* model subclasses. Doing this will give us the same functionality but allow us to easily customize our *User* model down the road.

## Project Level URL Routing

We’ll be building the features in this order:

URLs → views → serializers → models

This means we’ll be referencing some files before we make them, which may feel strange, but I feel doing it this way is more intuitive because it follows the path of the HTTP request more closely. When the request first comes into our server, we need to decide where to route that request. Kind of like a receptionist to our web server, our top-most URL routes will route the request to the proper modules and views. We’ll use the innermost auth\_api folder to store the top-most url routes along with our project-wide settings:

### auth\_api/urls.py

**from** **django.conf.urls** **import** url, include

**from** **django.contrib** **import** admin

**from** **rest\_framework.documentation** **import** include\_docs\_urls

**from** **auth\_api** **import** views

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'^docs/', include\_docs\_urls(title='Todo API', description='RESTful API for Todo')),

url(r'^$', views.api\_root),

url(r'^', include('users.urls', namespace='users')),

url(r'^', include('todos.urls', namespace='todos')),

]

When parsing the incoming request, Django uses regex to match the URL to the urlpatterns and forward the request to the place we want, which contains more URL routes or a view. We set up routes for our Django admin, documentation (automatically generated thanks to DRF), routes for users, routes for todos, and our api root.

## Making the Root View of the API

One of the routes we defined earlier routed to views.api\_root. Though not necessary, making a root view that acts as a table of contents to other routes in your api when requests matches your domain name exactly is easy to implement and improves developer experience. The urlpatterns we wrote will route requests, and the views we will write will handle those requests and return HTTP responses. To write our api\_root, we can use DRF’s built in decorator, @api\_view*,*to wrap our view with some nice utilities:

* Specifies which HTTP methods we allow the view to respond to
* Wraps normal HTTP request and response objects to provide a more uniform interface to work with HTTP data.

### auth\_api/views.py

**from** **rest\_framework.decorators** **import** api\_view

**from** **rest\_framework.response** **import** Response

**from** **rest\_framework.reverse** **import** reverse

@api\_view(['GET'])

**def** api\_root(request, format=None):

**return** Response({

'users': reverse('users:user-list', request=request, format=format),

'todos': reverse('todos:todo-list', request=request, format=format),

})

## Configuring URLs for Users and Todos

We need to create routes for when the request matches */todos/\** or */users/\** (as per the table we made earlier) and route them to the proper views to be handled:

### todos/urls.py

**from** **django.conf.urls** **import** url

**from** **rest\_framework.urlpatterns** **import** format\_suffix\_patterns

**from** **todos** **import** views

urlpatterns = [

url(r'^todos/$', views.TodoList.as\_view(), name='todo-list'),

url(r'^todos/(?P<pk>[0-9]+)/$', views.TodoDetail.as\_view(), name='todo-detail'),

]

### users/urls.py

**from** **django.conf.urls** **import** url

**from** **rest\_framework.urlpatterns** **import** format\_suffix\_patterns

**from** **users** **import** views

urlpatterns = [

url(r'^users/$', views.UserList.as\_view(), name='user-list'),

url(r'^users/(?P<pk>[0-9]+)/$', views.UserDetail.as\_view(), name='user-detail'),

]

## Writing the Views

Views handle the request and return a response, and by handling, I mean we can do anything we want with it — from hitting the database, modifying the request, structuring the response, or injecting our own logic into it. With respect to APIs, all the core views often do the same thing conceptually:

* Parse the request for data and the HTTP method
* Query the database (DB) to fetch the model object(s), if needed
* Serialize the data (we’ll discuss this more later)
* Do something with the object/data (create, read, update, delete)
* Return an HTTP response

Views are most often written as functions. Example of a function-based view that handles our API request looks something like this:

@api\_view(['GET', 'POST'])

**def** snippet\_list(request):

*"""*

*List all snippets, or create a new snippet.*

*"""*

**if** request.method == 'GET':

snippets = Snippet.objects.all()

serializer = SnippetSerializer(snippets, many=True)

**return** Response(serializer.data)

**elif** request.method == 'POST':

serializer = SnippetSerializer(data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data, status=status.HTTP\_201\_CREATED)

**return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

@api\_view(['GET', 'PUT', 'DELETE'])

**def** snippet\_detail(request, pk):

*"""*

*Retrieve, update or delete a snippet instance.*

*"""*

**try**:

snippet = Snippet.objects.get(pk=pk)

**except** Snippet.DoesNotExist:

**return** Response(status=status.HTTP\_404\_NOT\_FOUND)

**if** request.method == 'GET':

serializer = SnippetSerializer(snippet)

**return** Response(serializer.data)

**elif** request.method == 'PUT':

serializer = SnippetSerializer(snippet, data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data)

**return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**elif** request.method == 'DELETE':

snippet.delete()

**return** Response(status=status.HTTP\_204\_NO\_CONTENT)

Ref: <http://www.django-rest-framework.org/tutorial/2-requests-and-responses/>

So again, we see that the things that views essentially do are:

* Parse the request for data and the HTTP method
* Query the database (DB) to fetch the model object(s), if needed
* Serialize the data (we’ll discuss this more later)
* Do something with the object/data (create, read, update, delete)
* Return an HTTP response

It’s relatively straight-forward and explicit, but imagine having to write our views likes this for:

* Todo\_detail
* Todo\_list
* User\_detail
* User\_list

There would be a lot of logic duplication. Instead, we could use an object-oriented approach and use classes and inheritance to reuse common blocks of logic. Using class-based views, we could write our views like this:

### todos/views.py:

**from** **todos.models** **import** Todo

**from** **rest\_framework** **import** generics

**from** **rest\_framework.response** **import** Response

**from** **rest\_framework.reverse** **import** reverse

**from** **todos.serializers** **import** TodoSerializer

**class** TodoList(generics.ListCreateAPIView):

queryset = Todo.objects.all()

serializer\_class = TodoSerializer

**def** perform\_create(self, serializer):

serializer.save(user=self.request.user)

**class** TodoDetail(generics.RetrieveUpdateDestroyAPIView):

serializer\_class = TodoSerializer

**def** get\_queryset(self):

**return** Todo.objects.all().filter(user=self.request.user)

With that, we accomplish the exact same functionality as the example above that uses function-based views, only with significantly less code. In the above classes, we subclass generic classes provided by DRF. The generic classes provide methods and properties that encapsulate the common logic so we don’t have to keep rewriting it, but we can still access, overwrite, and customize the logic if needed. Blocks of logic like the ones that parse the DB for the correct model instance, or carry out error handling, or determining which serializer to use and how to instantiate it, or structure the response, or parsing the request, are already written for you and are readily available to the classes that you write if you subclass the generic ones. There is honestly a lot of cleverly written code hidden behind these generic views with a lot of concepts and techniques to learn from that are beyond the scope of this guide, but the best way to really learn and understand is to download the source code, read it, and tinker. Reading the documentation on these topics:

<https://docs.djangoproject.com/en/1.11/topics/class-based-views/>

<http://www.django-rest-framework.org/api-guide/generic-views/>

<https://github.com/encode/django-rest-framework/tree/master>

<https://github.com/django/django>

Try not to overthink it too much and remember that the 10 or so lines of code using class-based views does the exact same thing as the many more lines of codes using function-based views in the above example, just more concise. Repeat for Users:

### users/views.py

**from** **users.models** **import** User

**from** **rest\_framework** **import** generics

**from** **rest\_framework.response** **import** Response

**from** **rest\_framework.reverse** **import** reverse

**from** **users.serializers** **import** UserSerializer

**class** UserList(generics.ListCreateAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

**class** UserDetail(generics.RetrieveUpdateDestroyAPIView):

serializer\_class = UserSerializer

**def** get\_queryset(self):

**return** User.objects.all().filter(username=self.request.user)

## Writing the Serializers

**DRF serializers** provide the service of serialization and deserialization. **Serialization** is the process of translating data structures into a format that can be stored, which in this case means turning querysets and model instances into native Python datatypes and then into JSON. **Deserialization** is the opposite, taking JSON and turning it into native Python datatypes and then into model instances. If you’ve made an API in any language then you’ve used the same concepts serializers use. Serializers just give you a convenient interface to take data in one form and convert it into another. DRF serializers use an interface similar to that of Django forms in that you define the fields of the model you wish to serialize/deserialize and when you instantiate the serializer with data it will do the field validation for you.

<http://www.django-rest-framework.org/api-guide/serializers/>

Let’s write our serializers for Todos and Users:

### Todos/serializers.py

**from** **rest\_framework** **import** serializers

**from** **todos.models** **import** Todo

**class** TodoSerializer( serializers.HyperlinkedModelSerializer):

user = serializers.ReadOnlyField( source='user.username')

**class** Meta:

model = Todo

fields = ('url', 'id', 'created', 'name', 'user')

extra\_kwargs = {

'url': {

'view\_name': 'todos:todo-detail',

}

}

### users/serializers.py

**from** **rest\_framework** **import** serializers

**from** **users.models** **import** User

**class** UserSerializer( serializers.HyperlinkedModelSerializer):

todos = serializers.HyperlinkedRelatedField(

many=True,

view\_name='todos:todo-detail',

read\_only=True

)

password = serializers.CharField(write\_only=True)

**def** create(self, validated\_data):

user = User(

username=validated\_data.get('username', None)

)

user.set\_password(validated\_data.get('password', None))

user.save()

**return** user

**def** update(self, instance, validated\_data):

**for** field **in** validated\_data:

**if** field == 'password':

instance.set\_password (validated\_data.get(field))

**else**:

instance.\_\_setattr\_\_(field, validated\_data.get(field))

instance.save()

**return** instance

**class** Meta:

model = User

fields = ('url', 'id', 'username',

'password', 'first\_name', 'last\_name',

'email', 'todos'

)

extra\_kwargs = {

'url': {

'view\_name': 'users:user-detail',

}

}

Notice that *UserSerializer* is more complex than *TodoSerializer*. Serializers, like our class-based views, come in many varieties and can inherit from many generic base classes. Within those base classes are methods and properties we can overwrite to customize at certain hooks, like when a serializer updates or saves data. We customize *UserSerializer* because we want to use the hashing feature that comes with Django’s *AbstractUser* when updating and saving users for security reasons.

## Writing the Models

From client to url routes to views to serializers, and now to the last part in our little app, the models. Our models are what define our Data. Django models get written in normal Python classes and they get mapped to SQL DBs, each attribute on the model getting mapped to a field in its respective table.

<https://docs.djangoproject.com/en/1.11/topics/db/models/>

We’ll create two models, Todo and User, and we’ll create a many-to-one relationship (foreign key) from our Todos to our Users, so a User can have many Todos but a Todo will only have one User.

### todos/models.py

**from** **django.db** **import** models

**from** **users.models** **import** User

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

created = models.DateTimeField(auto\_now\_add=True)

name = models.CharField(max\_length=100, unique=True, blank=False, null=False)

user = models.ForeignKey('users.User', related\_name='todos', on\_delete=models.CASCADE, null=False)

**class** Meta:

ordering = ('created',)

### users/models.py

**from** **django.db** **import** models

**from** **django.contrib.auth.models** **import** AbstractUser

**class** User(AbstractUser):

**pass**

We touched a bit on this earlier, but we don’t do anything extra with our *User*model. We use the same *AbstractUser* that the default Django *User* model uses, but defining our own *User* model gives us the ability to easily edit it later on while still retaining the same features as the default Django *User.*

## Bringing It All Together

We’ve got all the code we need to make this API work, and now we just need to migrate our models and create a superuser.

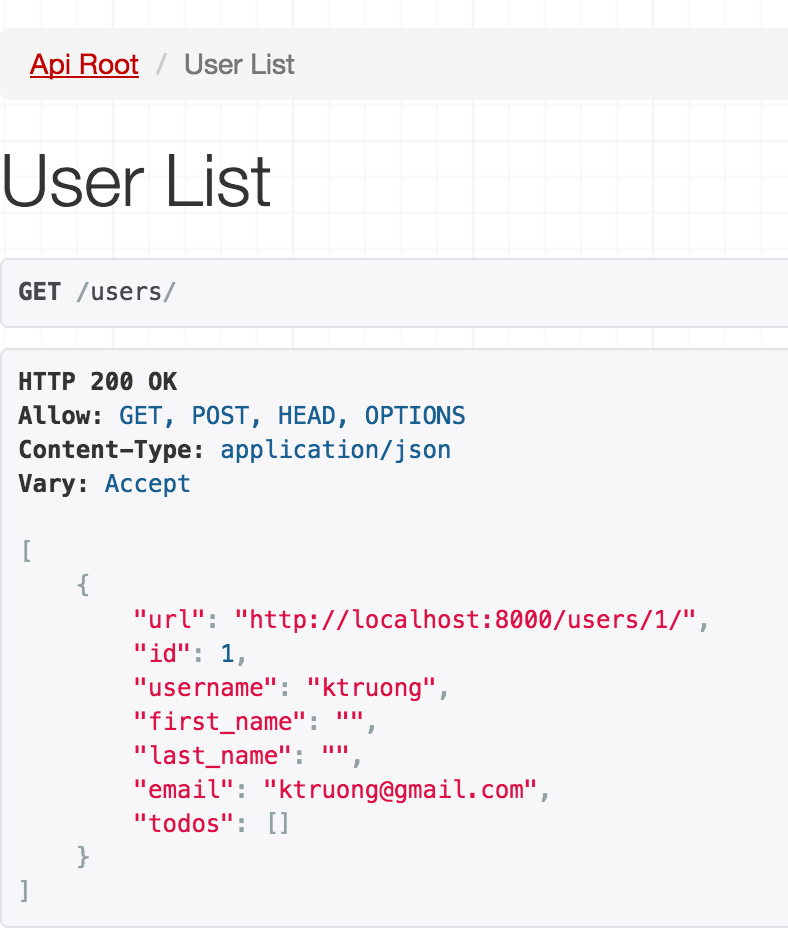
ktruong$ python manage.py makemigrations todos users

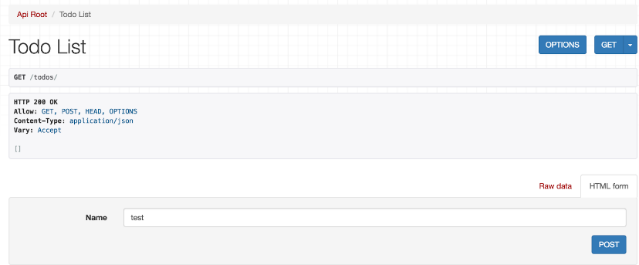
ktruong$ python manage.py migrate

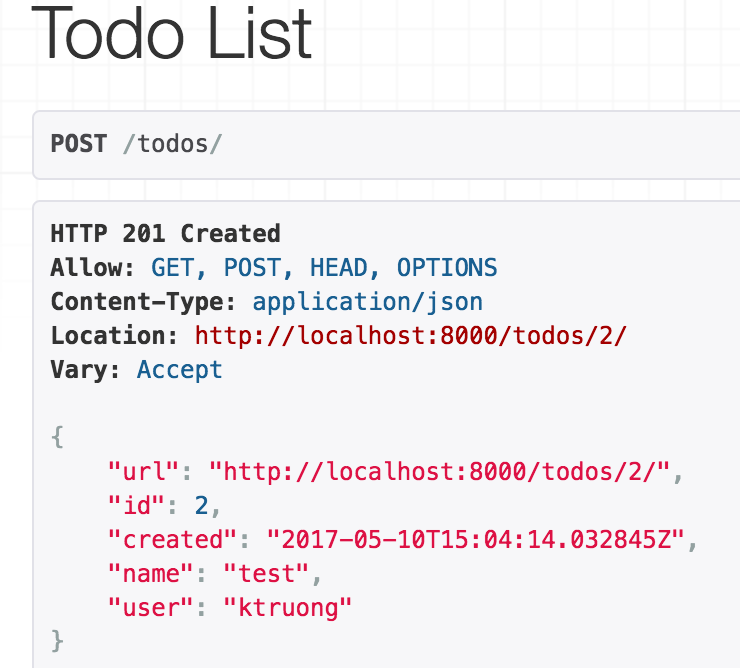
ktruong$ python manage.py createsuperuser

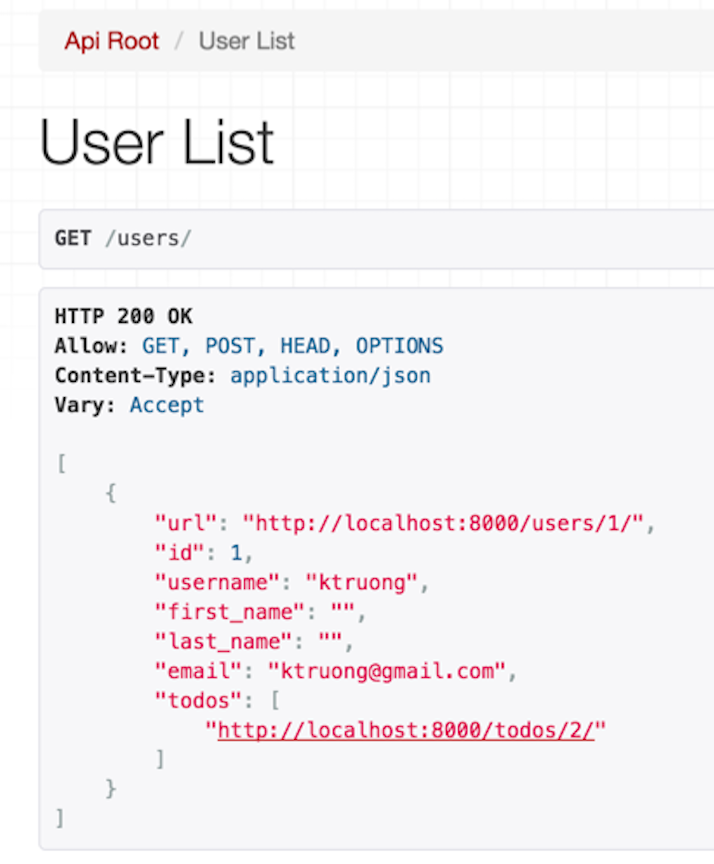
ktruong$ python manage.py runserver

Now go browse localhost:8000, login to the admin if needed, and play around with your new API. The browsable API is just another out-of-the-box feature from DRF and it allows you to browse your API interactively in a browser. You can click through links, see relationships, see models and objects, perform any CRUD (create, read, update, delete) request, and do pretty much anything you can do with an API, but in a visible and interactive way.









## Conclusion

We covered some fundamental concepts of building a RESTful API by creating a basic Todo API, with complete CRUD endpoints. Though very barebones, the instructions in this guide should serve as another step in the ladder to help you become a better developer and understand how to make your own API. There’s still a lot more one can do to improve your API, such as adding authentication and authorization with JSON Web Tokens, adding unit tests, and more, which are also topics I plan on covering in the future.

# Django REST: Beginners Guide

Nov 14, 2018

This is a beginner-friendly guide to the official Django Rest Framework [tutorial](http://www.django-rest-framework.org/tutorial/1-serialization/). We will build the exact same code-highlighting web API but include detailed explanations and complete code samples along the way. However we’ll skip sections on the Django shell, function-based views, and other areas that are helpful but also typically confusing for newcomers. Complete source code is [available on Github](https://github.com/wsvincent/rest-framework-tutorial). **NOTE**: This tutorial is powered by the book [Django for APIs](https://djangoforapis.com/) which covers how to build and test multiple web APIs with Django and Django REST Framework.

## Initial Setup

Go to [Django For Beginners](https://djangoforbeginners.com/initial-setup/) for a guide to properly installing Python 3.7 and [pipenv](https://docs.pipenv.org/) on Mac, Windows, or Linux computer. Then create a virtual environment for our project that contains django, djangorestframework, and pygments which is used for code highlighting. Then ativate the virtual environment.

$ pipenv install django djangorestframework pygments

$ pipenv shell

(env) $

**Note**: The name of your virtual environment is listed as (env) . In practice, the name will be some variation of your current directory name. To exit at any time, type exit. Let’s create a new project to work with called tutorial and an app within it called snippets for our web API. Don’t forget that period . at the end or Django will add an extra directory for us that we don’t need!

(env) $ django-admin.py startproject tutorial .

(env) $ python manage.py startapp snippets

Add snippets app and rest\_framework to INSTALLED\_APPS config in our tutorial/settings.py file.

*# tutorial/settings.py*

INSTALLED\_APPS **=** [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework', *# new*

'snippets', *# new*

]

## Models

The model is a good place to start any new project. In the snippets/models.pyfile, create a new model called Snippet.

*# snippets/models.py*

from django.db import models

from pygments.lexers import get\_all\_lexers

from pygments.styles import get\_all\_styles

LEXERS **=** [item **for** item **in** get\_all\_lexers() **if** item[1]]

LANGUAGE\_CHOICES **=** sorted([(item[1][0], item[0]) **for** item **in** LEXERS])

STYLE\_CHOICES **=** sorted((item, item) **for** item **in** get\_all\_styles())

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

created **=** models**.**DateTimeField(auto\_now\_add**=**True)

title **=** models**.**CharField(max\_length**=**100, blank**=**True, default**=**'')

code **=** models**.**TextField()

linenos **=** models**.**BooleanField(default**=**False)

language **=** models**.**CharField(choices**=**LANGUAGE\_CHOICES, default**=**'python', max\_length**=**100)

style **=** models**.**CharField(choices**=**STYLE\_CHOICES, default**=**'friendly', max\_length**=**100)

**class** **Meta**:

ordering **=** ('created',)

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

**return** self**.**title

Then create an initial migration file and sync the database.

(env) $ python manage.py makemigrations snippets

(env) $ python manage.py migrate

We need to add some data into our model to make it “real” now. The official tutorial uses the Django shell for this however the Django admin is a more intuitive, visual approach for many developers. We’ll use that instead. First however we need to update snippets/admin.py so the app will appear in the admin.

*# snippets/admin.py*

from django.contrib import admin

from . models import Snippet

admin**.**site**.**register(Snippet)

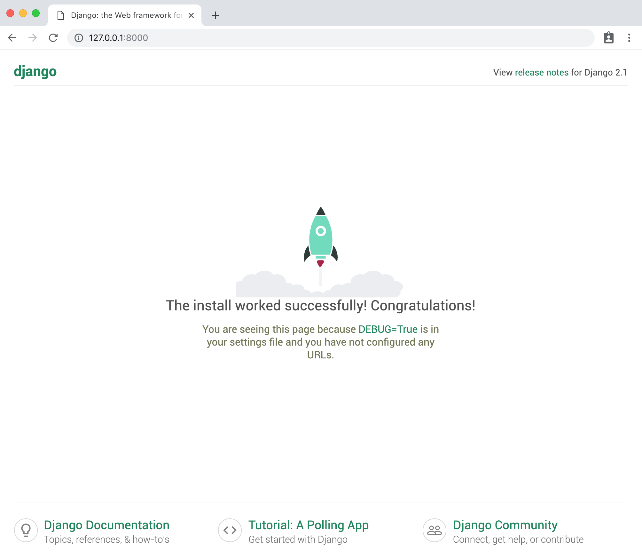
Then create a superuser account we can login with. Follow all the prompts that follow the command below for setting a username, email, and password.

(env) $ python manage.py createsuperuser

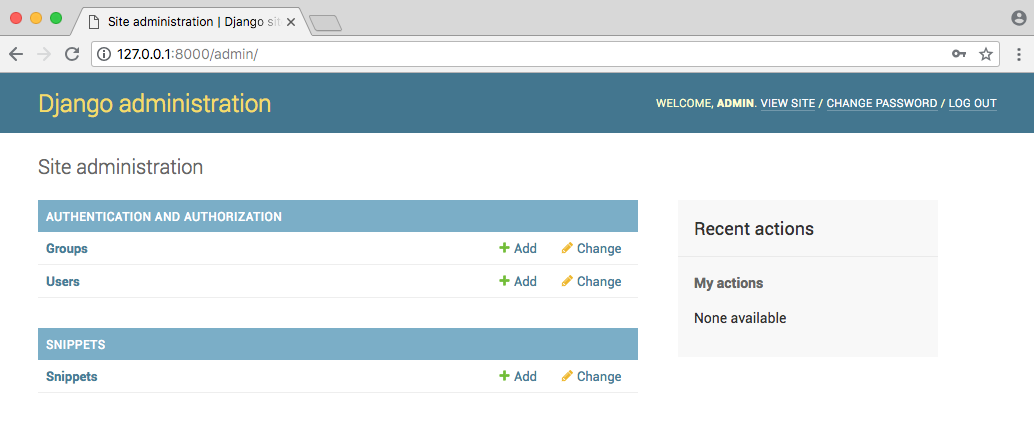
Now start our local web server for the first time.

(env) $ python manage.py runserver

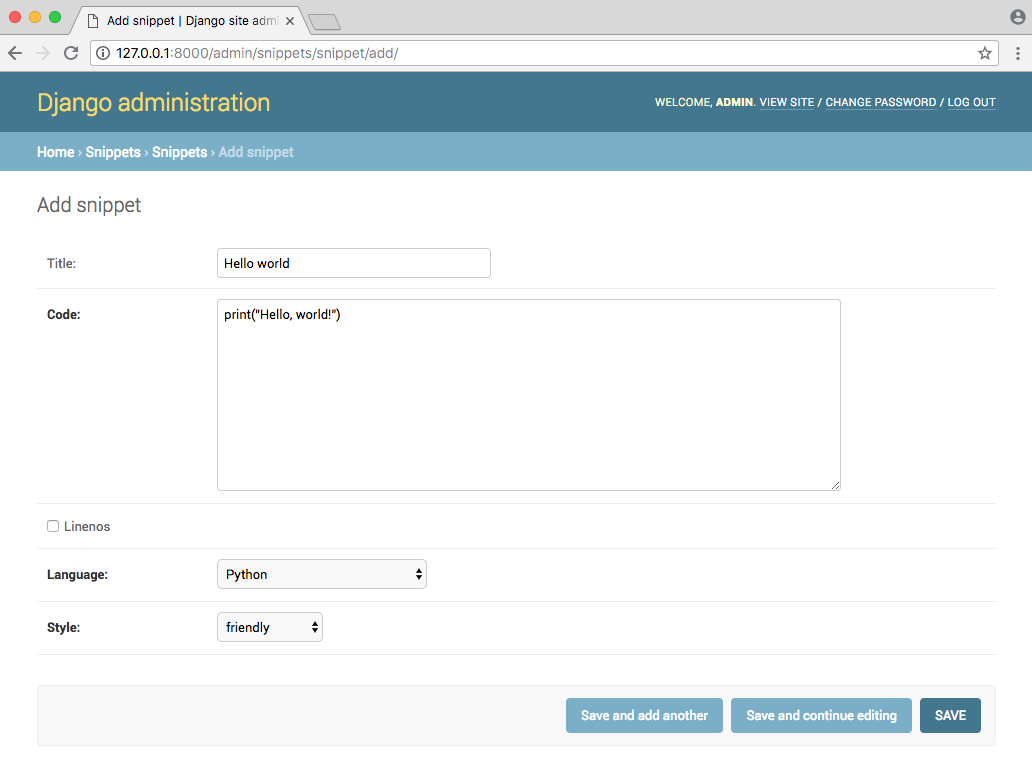
Now navigate to Django homepage at <http://127.0.0.1:8000/> to confirm everything is working.

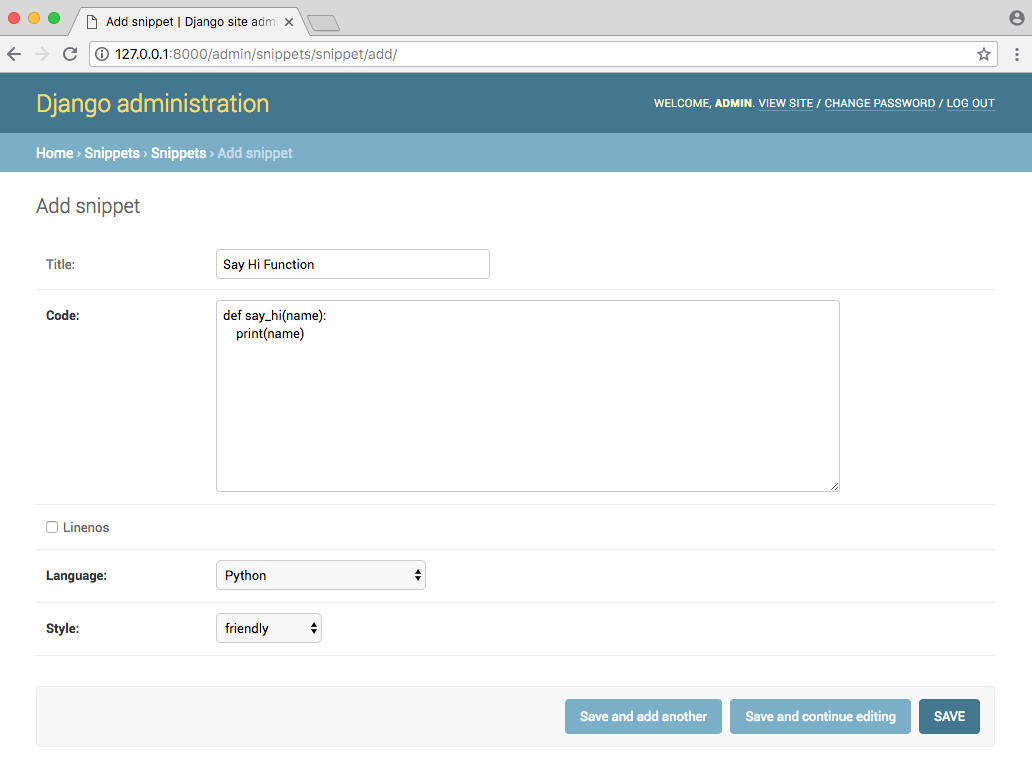


Then switch to Django admin at <http://127.0.0.1:8000/admin/>. Log in with your superuser account.

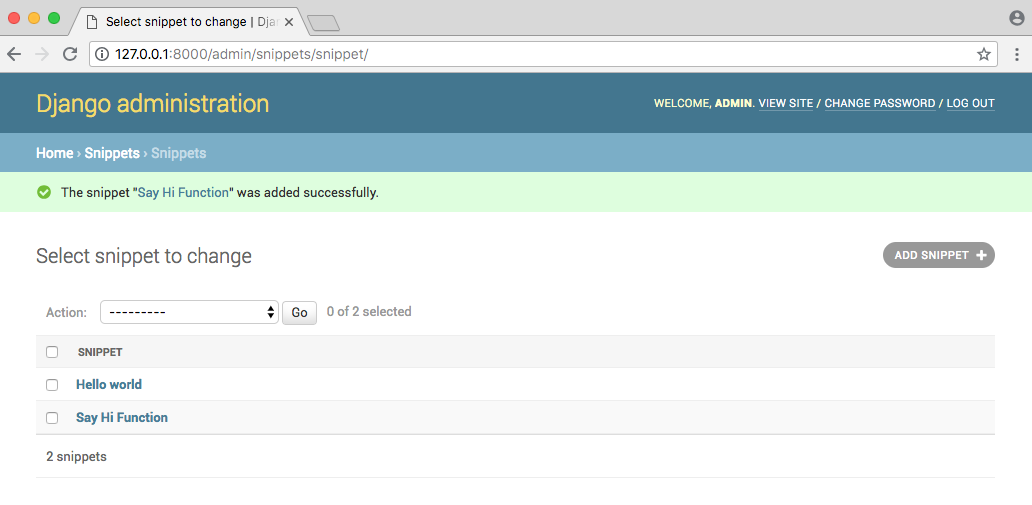


Click “+ Add” button next to Snippets. Create 2 new snippets.





Click the “Save” button in the lower right for each snippet. Both will be visible on the main Snippets page.



## Serialization

A *Serializer* transforms model instances into JSON. This the real “magic” that Django Rest Framework provides for us since ultimately a web API endpoint returns JSON and available HTTP verbs. Create a snippets/serializers.py file.

(env) $ touch snippets/serializers.py

Extend DRF’s [ModelSerializer](http://www.django-rest-framework.org/api-guide/serializers/#modelserializer) to create a SnippetSerializer class that uses our model and outputs the table fields.

*# snippets/serializers*

from rest\_framework import serializers

from .models import Snippet, LANGUAGE\_CHOICES, STYLE\_CHOICES

**class** **SnippetSerializer**(serializers**.**ModelSerializer):

**class** **Meta**:

model **=** Snippet

fields **=** ('id', 'title', 'code', 'linenos',

'language', 'style', )

Next we need a view that handles the logic of combining a model, serializer, and eventually URL together. Just as traditional Django ships with several class-based generic views to handle common functionality, so too Django Rest Framework has its own set of powerful class-based generic views we can use. Specifically use [ListCreateAPIView](http://www.django-rest-framework.org/api-guide/generic-views/#listcreateapiview) to create read-only endpoint that lists all available Snippet instances; then [RetrieveUpdateDestroyAPIView](http://www.django-rest-framework.org/api-guide/generic-views/#retrieveupdatedestroyapiview) for a detail view of snippets to support CRUD-like functionality.

*# snippets/views.py*

from rest\_framework import generics

from .models import Snippet

from .serializers import SnippetSerializer

**class** **SnippetList**(generics**.**ListCreateAPIView):

queryset **=** Snippet**.**objects**.**all()

serializer\_class **=** SnippetSerializer

**class** **SnippetDetail**(generics**.**RetrieveUpdateDestroyAPIView):

queryset **=** Snippet**.**objects**.**all()

serializer\_class **=** SnippetSerializer

## URLs

Final step: configure URLs. Topmost, project-level tutorial/urls.py file, add include as an import for the snippets app urls which appears at the empty string ''.

*# tutorial/urls.py*

from django.contrib import admin

from django.urls import include, path *# new*

urlpatterns **=** [

path('admin/', admin**.**site**.**urls),

path('', include('snippets.urls')), *# new*

]

Then create a urls.py file with our snippets app.

(env) $ touch snippets/urls.py

And add the following code.

*# snippets/urls.py*

from django.urls import path

from rest\_framework.urlpatterns import format\_suffix\_patterns

from snippets import views

urlpatterns **=** [

path('snippets/', views**.**SnippetList**.**as\_view()),

path('snippets/<int:pk>/', views**.**SnippetDetail**.**as\_view()),

]

urlpatterns **=** format\_suffix\_patterns(urlpatterns)

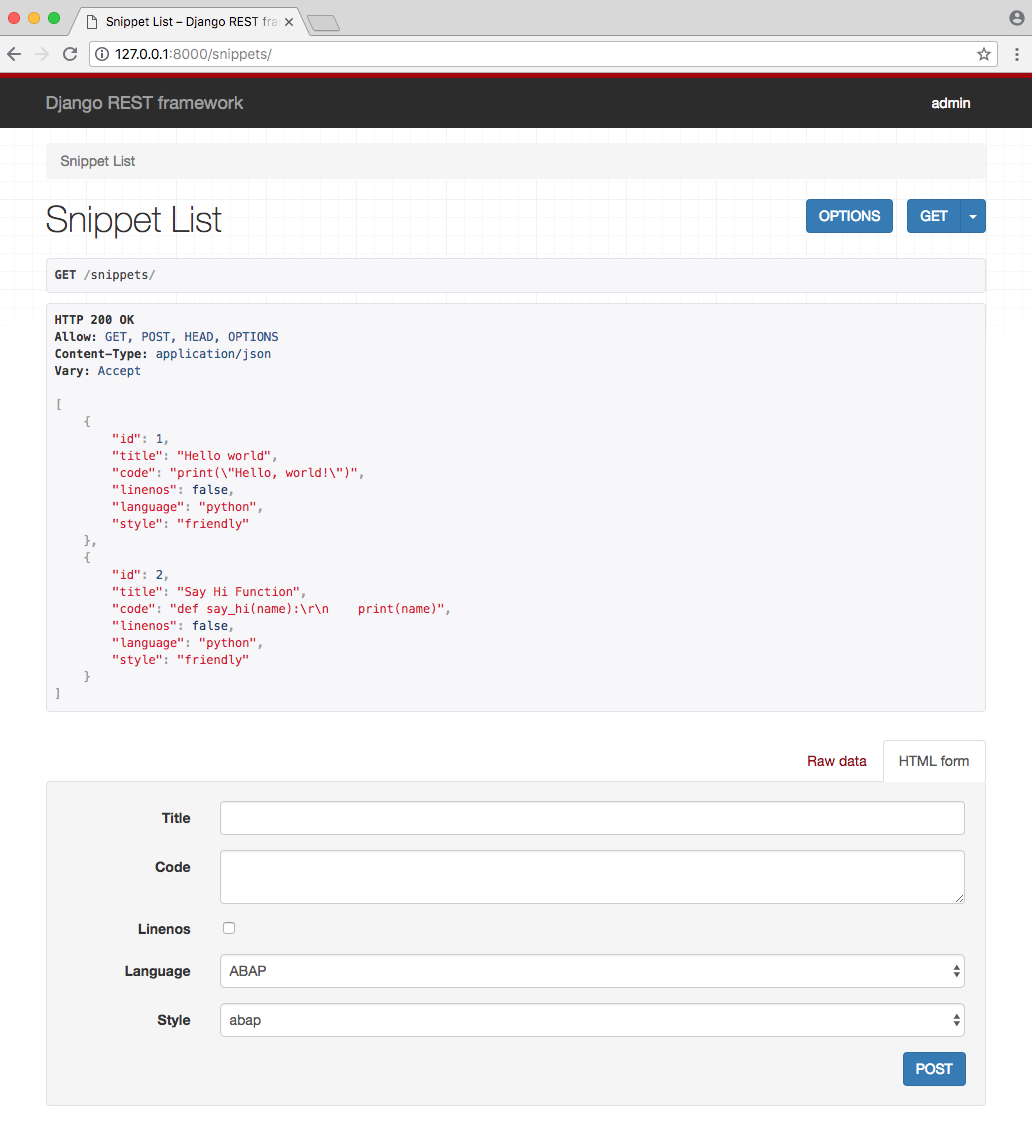
Including [format\_suffix\_patterns](http://www.django-rest-framework.org/api-guide/format-suffixes/#format_suffix_patterns) is an optional choice that provides a simple, DRY way to refer to a specific file format for a URL endpoint. It means our API will be able to handle URls such as <http://example.com/api/items/4.json> rather than just <http://example.com/api/items/4>.

## Browsable API

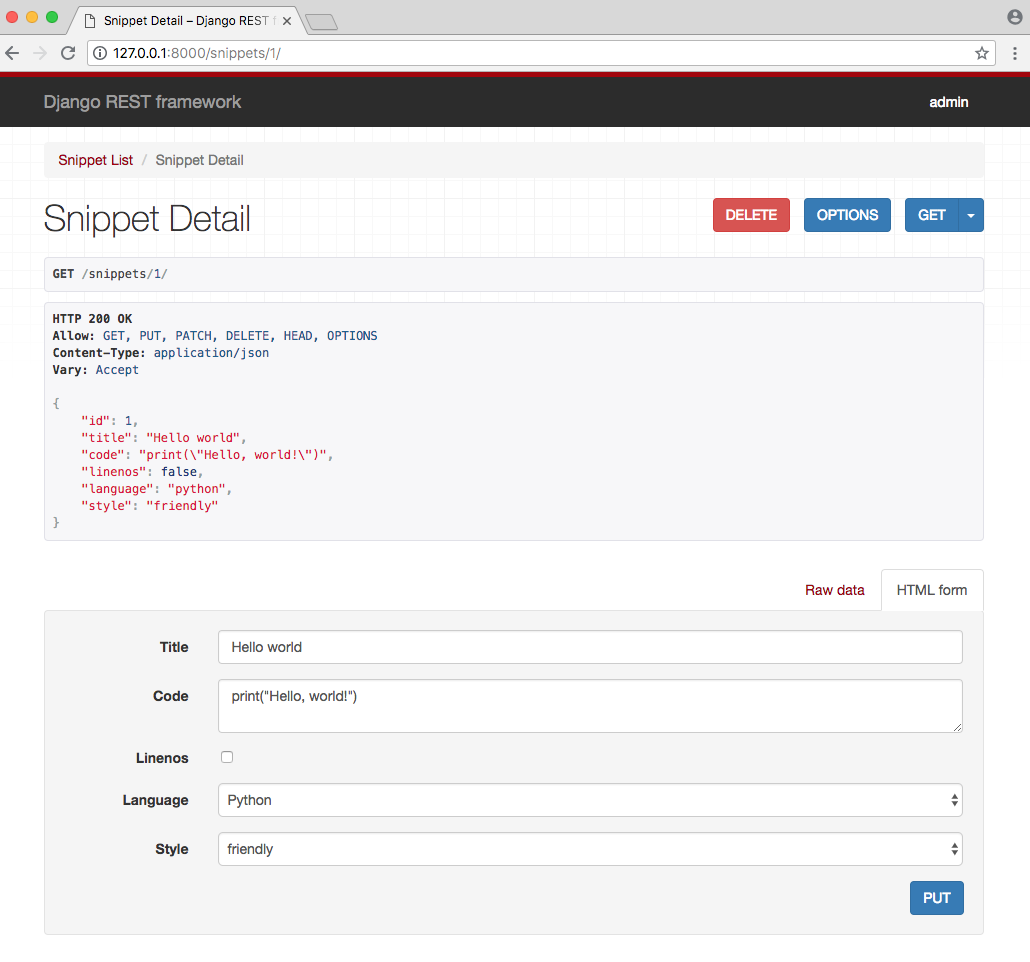
Django Rest Framework ships with a browsable API that we can now use. Make sure the local server is running.

(env) $ python manage.py runserver

Navigate to Snippets List endpoint: <http://127.0.0.1:8000/snippets/>.



We can also go to the detail view for each snippet. For example, the first snippet is at <http://127.0.0.1:8000/snippets/1/>.



Reminder: the id is automatically set by Django on database entry.

## Requests and Responses

Currently our API has no restrictions on who can edit or delete code snippets. In this section we will make sure that:

* Code snippets are always associated with a creator
* Only authenticated users may create snippets
* Only the creator of a snippet may update or delete it
* Unauthenticated requests should have full read-only access

## Adding information to our model

First up let’s add two fields to our existing Snippet model class: owner which will represent the user who created the code snippet and highlighted to store the highlighted HTML representation of the code. We also want to ensure that when the model is saved, we use the pygments code highlighting library to populate our highlighted field. So we’ll need some additional imports as well as a .save() method.

*# snippets/models.py*

from django.db import models

from pygments import highlight *# new*

from pygments.formatters.html import HtmlFormatter *# new*

from pygments.lexers import get\_all\_lexers, get\_lexer\_by\_name*#new*

from pygments.styles import get\_all\_styles

LEXERS **=** [item **for** item **in** get\_all\_lexers() **if** item[1]]

LANGUAGE\_CHOICES **=** sorted([(item[1][0], item[0]) **for** item **in** LEXERS])

STYLE\_CHOICES **=** sorted((item, item) **for** item **in** get\_all\_styles())

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

created **=** models**.**DateTimeField(auto\_now\_add**=**True)

title **=** models**.**CharField(max\_length**=**100, blank**=**True, default**=**'')

code **=** models**.**TextField()

linenos **=** models**.**BooleanField(default**=**False)

language **=** models**.**CharField(choices**=**LANGUAGE\_CHOICES, default**=**'python', max\_length**=**100)

style **=** models**.**CharField(choices**=**STYLE\_CHOICES, default**=**'friendly', max\_length**=**100)

owner **=** models**.**ForeignKey('auth.User', related\_name**=**'snippets', on\_delete**=**models**.**CASCADE) *# new*

highlighted **=** models**.**TextField() *# new*

**class** **Meta**:

ordering **=** ('created',)

**def** **save**(self, **\***args, **\*\***kwargs): *# new*

"""

Use the `pygments` library to create a highlighted HTML

representation of the code snippet.

"""

lexer **=** get\_lexer\_by\_name(self**.**language)

linenos **=** 'table' **if** self**.**linenos **else** False

options **=** {'title': self**.**title} **if** self**.**title **else** {}

formatter **=** HtmlFormatter(style**=**self**.**style, linenos**=**linenos,

full**=**True, **\*\***options)

self**.**highlighted **=** highlight(self**.**code, lexer, formatter)

super(Snippet, self)**.**save(**\***args, **\*\***kwargs)

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

**return** self**.**title

Normally we would create a migration and sync it to update our tables. However since we have added an owner here and have existing content, it’s simpler to just delete the database and start again. Make sure you have stopped the local server with Control+c.

(env) $ rm -f db.sqlite3

(env) $ rm -r snippets/migrations

(env) $ python manage.py makemigrations snippets

(env) $ python manage.py migrate

Re-create our steps from earlier to create a new superuser account. We’ll want a second superuser account which is simplest to setup from the command line too. So run createsuperuser twice. I’ve called my users admin and testuser. Then start up the local server.

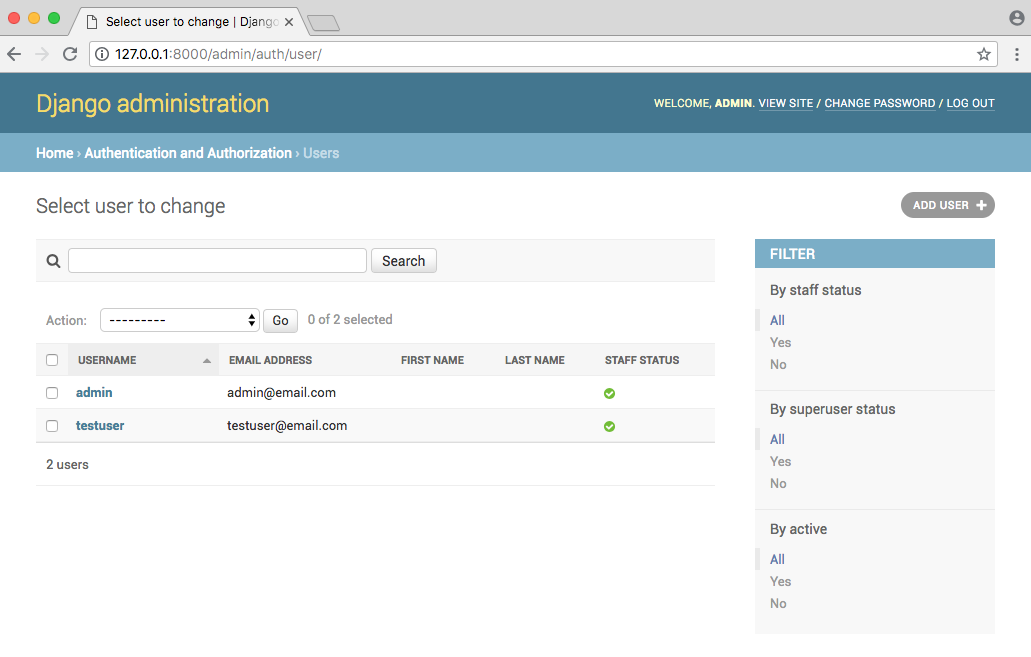
(env) $ python manage.py createsuperuser

(env) $ python manage.py createsuperuser

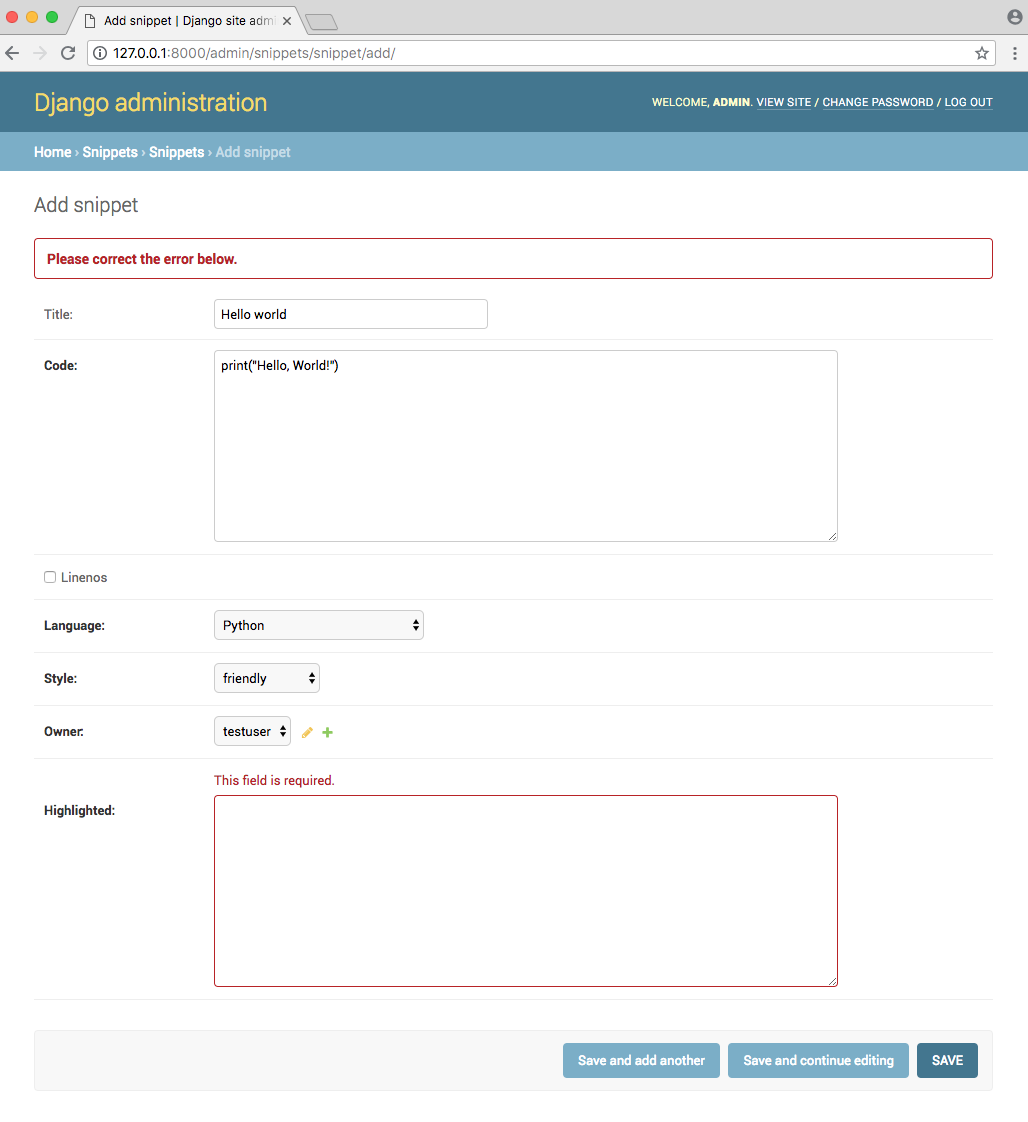
(env) $ python manage.py runserver

Go back to Django admin at <http://127.0.0.1:8000/admin/> and login with the admin account.

If you click on the Users link you will be redirected to the Users page which should show both users. Once complete, you should see the two users on the Users page.



We need to recreate our snippets too since the initial database was destroyed. Create a new snippet and specify Owner as one of our users. Choose testuser. But there’s a problem when we “Save”.



We get a ValidationError here. In the tutorial the Django shell is used to input data, but we used the admin here. So the existing code doesn’t work as is. Recall that the highlighted field is automatically set by our custom save() method on the model, but the admin doesn’t know this. It expects us to enter a value. To solve, update our admin.py file and set highlighted as a read-only field.

*# snippets/admin.py*

from django.contrib import admin

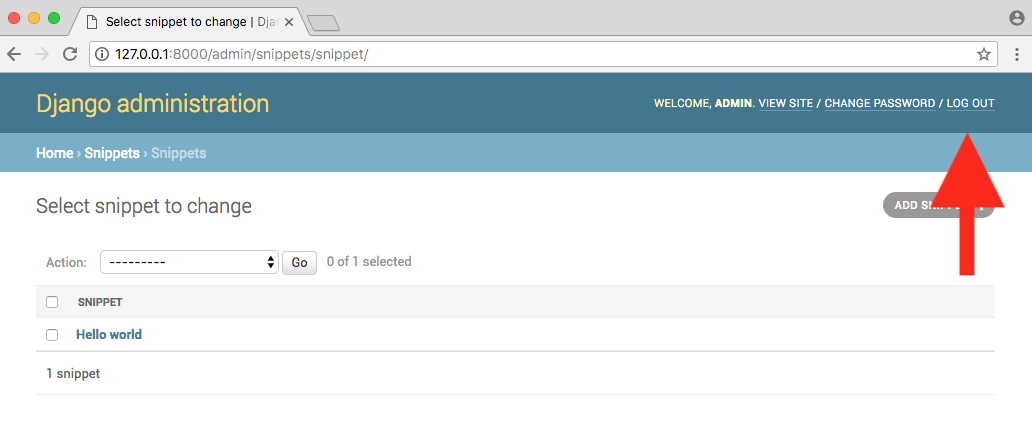
from . models import Snippet

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

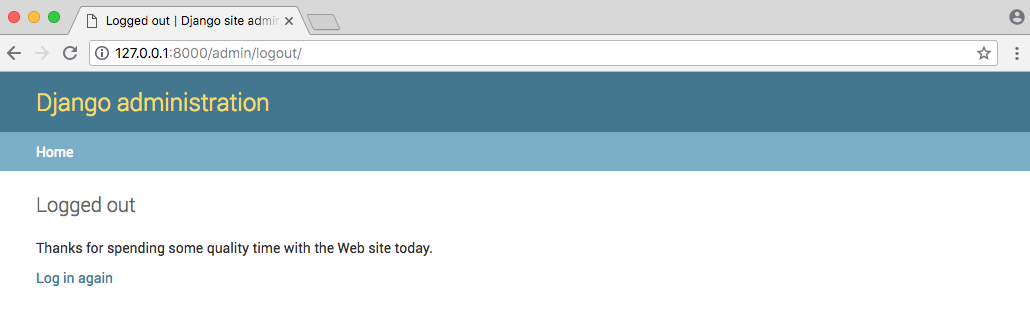
readonly\_fields **=** ('highlighted',)

admin**.**site**.**register(Snippet, SnippetAdmin)

Try clicking the “Save” button again. It should work. Final step: click the Log out link in the upper right corner of the admin page.



We will shortly be adding permissions to our API so that only authenticated (logged-in) users have access.



## Adding endpoints to our User models

Now let’s add endpoints for them to our API. Add a new UserSerializer class to the snippets/serializers.py file.

*# snippets/serializers.py*

from django.contrib.auth.models import User

from rest\_framework import serializers

from snippets.models import Snippet, LANGUAGE\_CHOICES, STYLE\_CHOICES

**class** **SnippetSerializer**(serializers**.**ModelSerializer):

**class** **Meta**:

model **=** Snippet

fields **=** ('id', 'title', 'code', 'linenos',

'language', 'style', )

**class** **UserSerializer**(serializers**.**ModelSerializer):

snippets **=** serializers**.**PrimaryKeyRelatedField(

many**=**True, queryset**=**Snippet**.**objects**.**all())

**class** **Meta**:

model **=** User

fields **=** ('id', 'username', 'snippets')

Because snippets is a *reverse* relationship on the default Django User model, it will not be included by default using the ModelSerializer class. We need to add an explicit field. We also need to add 2o new read-only views for a list of all users and a detail view of individual users. Note that we use the generic class-based RetrieveAPIView for the read-only detail view. And that we import both User and UserSerializer at the top.

*# snippets/views.py*

from django.contrib.auth.models import User *# new*

from rest\_framework import generics

from .models import Snippet

from .serializers import SnippetSerializer, UserSerializer *# new*

**class** **SnippetList**(generics**.**ListCreateAPIView):

queryset **=** Snippet**.**objects**.**all()

serializer\_class **=** SnippetSerializer

**class** **SnippetDetail**(generics**.**RetrieveUpdateDestroyAPIView):

queryset **=** Snippet**.**objects**.**all()

serializer\_class **=** SnippetSerializer

**class** **UserList**(generics**.**ListAPIView): *# new*

queryset **=** User**.**objects**.**all()

serializer\_class **=** UserSerializer

**class** **UserDetail**(generics**.**RetrieveAPIView): *# new*

queryset **=** User**.**objects**.**all()

serializer\_class **=** UserSerializer

Finally we need to add the new views to the API by configuring their URL routes. Add the following pattern to snippets/urls.py.

*# snippets/urls.py*

from django.urls import path

from rest\_framework.urlpatterns import format\_suffix\_patterns

from snippets import views

urlpatterns **=** [

path('snippets/', views**.**SnippetList**.**as\_view()),

path('snippets/<int:pk>/', views**.**SnippetDetail**.**as\_view()),

path('users/', views**.**UserList**.**as\_view()), *# new*

path('users/<int:pk>/', views**.**UserDetail**.**as\_view()), *# new*

]

urlpatterns **=** format\_suffix\_patterns(urlpatterns)

## Associating Snippets with Users

Currently there is no way to automatically associate the logged-in user that created a snippet with the snippet instance. We can set this automatically by overriding .perform\_create() method on our snippet views that lets us modify how an instance is saved. Add the following method to our existing SnippetList view class.

*# snippets/views.py*

**class** **SnippetList**(generics**.**ListCreateAPIView):

queryset **=** Snippet**.**objects**.**all()

serializer\_class **=** SnippetSerializer

**def** **perform\_create**(self, serializer): *# new*

serializer**.**save(owner**=**self**.**request**.**user)

## Updating our serializer

Now that snippets are associated with the user that created them, let’s update SnippetSerializer with an owner to reflect that. Make sure to also include owner in the list of fields too.

*# snippets/serializers.py*

from django.contrib.auth.models import User

from rest\_framework import serializers

from snippets.models import Snippet, LANGUAGE\_CHOICES, STYLE\_CHOICES

**class** **SnippetSerializer**(serializers**.**ModelSerializer):

owner **=** serializers**.**ReadOnlyField(source**=**'owner.username')*#new*

**class** **Meta**:

model **=** Snippet

fields **=** ('id', 'title', 'code', 'linenos',

'language', 'style', 'owner',) *# new*

**class** **UserSerializer**(serializers**.**ModelSerializer):

snippets **=** serializers**.**PrimaryKeyRelatedField(many**=**True, queryset**=**Snippet**.**objects**.**all())

**class** **Meta**:

model **=** User

fields **=** ('id', 'username', 'snippets')

The source argument used here controls which attribute is used to populate a field and can point to *any* attribute on the serialized instance. Also note that we’re using ReadOnlyField which is *always* read-only; it can not be used for updating model instances when they are serialized. We could use CharField(read\_only=True) here to accomplish the same thing.

## Adding required permissions to views

Now that code snippets are associated with users, we want to make sure that **only** authenticated users are able to create, update, and delete code snippets. Django Rest Framework ships with a number of permission classes we could use to restrict access to a given view. Here we will use IsAuthenticatedOrReadOnly to ensure that authenticated requests have read-write access and unauthenticated requests only have read-only access.

*# snippets/views.py*

from django.contrib.auth.models import User

from rest\_framework import generics, permissions *# new*

from .models import Snippet

from .serializers import SnippetSerializer, UserSerializer

**class** **SnippetList**(generics**.**ListCreateAPIView):

queryset **=** Snippet**.**objects**.**all()

serializer\_class **=** SnippetSerializer

permission\_classes **=** (permissions**.**IsAuthenticatedOrReadOnly,) *# new*

**def** **perform\_create**(self, serializer):

serializer**.**save(owner**=**self**.**request**.**user)

**class** **SnippetDetail** (generics**.**RetrieveUpdateDestroyAPIView):

queryset **=** Snippet**.**objects**.**all()

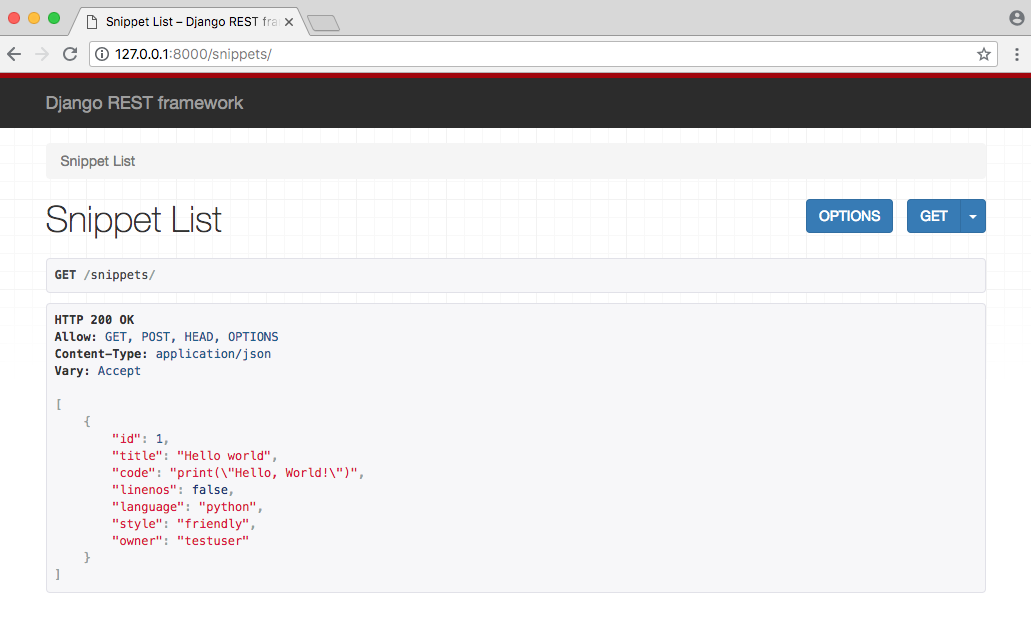
serializer\_class **=** SnippetSerializer

permission\_classes **=** (permissions**.**IsAuthenticatedOrReadOnly,) *# new*

**...**

## Adding login to the browsable API

Navigate to browsable API at <http://127.0.0.1:8000/snippets/>.



Since we are logged out, notice that you no longer CAN create new code snippets. to do so you need to be logged in as a user. We can add a login view to the browsable API by editing the URLconf in our project-level tutorial/urls.py file. Add rest\_framework.urls to the route api-auth/.

*# tutorial/urls.py*

from django.contrib import admin

from django.urls import include, path

urlpatterns **=** [

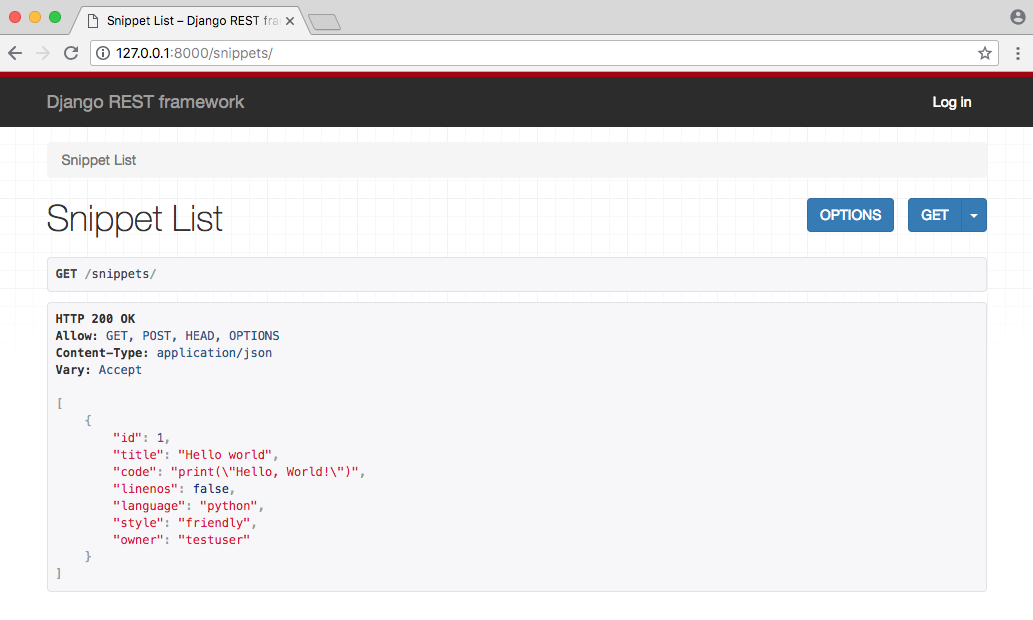
path('admin/', admin**.**site**.**urls),

path('api-auth/', include('rest\_framework.urls')),*#new*

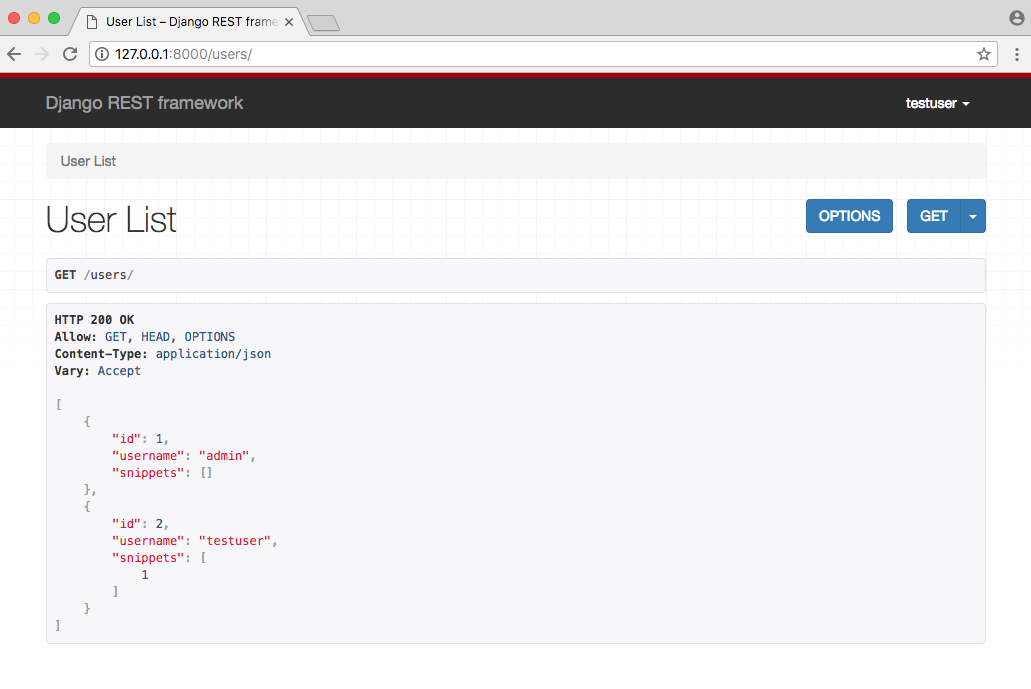
path('', include('snippets.urls')),

]

Note that the actual route used does not matter. Instead of api-auth/ we could also have used something-else/. The important thing is that rest\_framework.urls was included. Now open up the browser again and refresh the page. You will see a Log in link in the top right of the page.



Log in with testuser account. navigate to <http://127.0.0.1:8000/users/> endpoint. Notice that snipped ids are associated with each user.



We only have one snippet, made with our testuser account and containing the primary id of 1. If we added additional snippets for each user, they’d appear here as well. So things are working.

## Object level permissions

We’d like all code snippets to be visible to only the user that created a code snippet can update or delete it. Django Rest Framework gives us several options for setting permissions: at a project-level, view level, or object level. We will implement the last option and create a custom permission we can add to our SnippetDetail view class. Create a new permissions.py file.

(env) $ touch snippets/permissions.py

Then add the following code which extends Django Rest Framework’s existing permissions classes.

*# snippets/permissions.py*

from rest\_framework import permissions

**class** **IsOwnerOrReadOnly**(permissions**.**BasePermission):

"""

Custom permission to allow only owners of object to edit.

"""

**def** **has\_object\_permission**(self, request, view, obj):

*# Read permissions are allowed to any request,*

*# so we'll always allow GET, HEAD or OPTIONS requests.*

**if** request**.**method **in** permissions**.**SAFE\_METHODS:

**return** True

*# Write permissions only allowed to owner of snippet.*

**return** obj**.**owner **==** request**.**user

Next add the new custom permission to SnippetDetail by importing it at the top and including it in permission\_classes.

*# snippets/views.py*

from django.contrib.auth.models import User

from rest\_framework import generics, permissions

from .models import Snippet

from .permissions import IsOwnerOrReadOnly *# new*

from .serializers import SnippetSerializer, UserSerializer

**class** **SnippetDetail**(generics**.**RetrieveUpdateDestroyAPIView):

queryset **=** Snippet**.**objects**.**all()

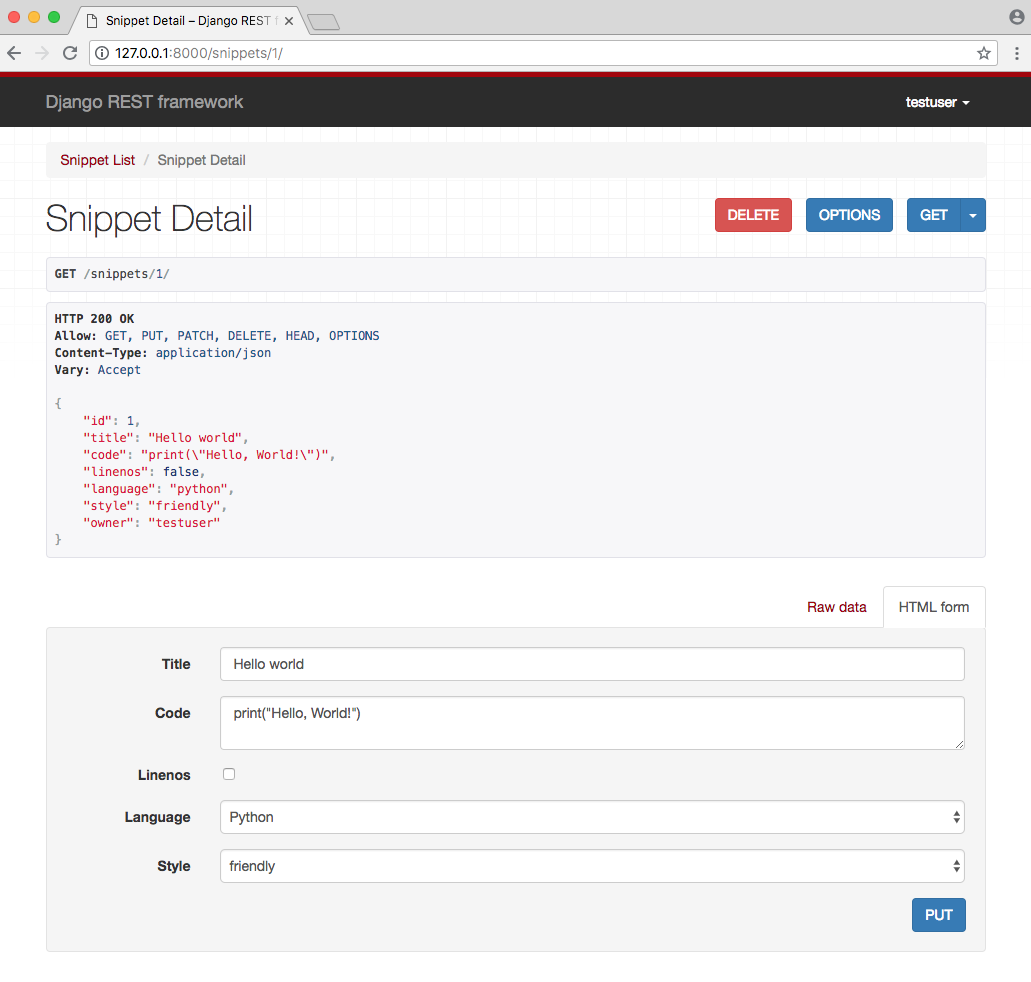
serializer\_class **=** SnippetSerializer

permission\_classes **=** (permissions**.**IsAuthenticatedOrReadOnly,

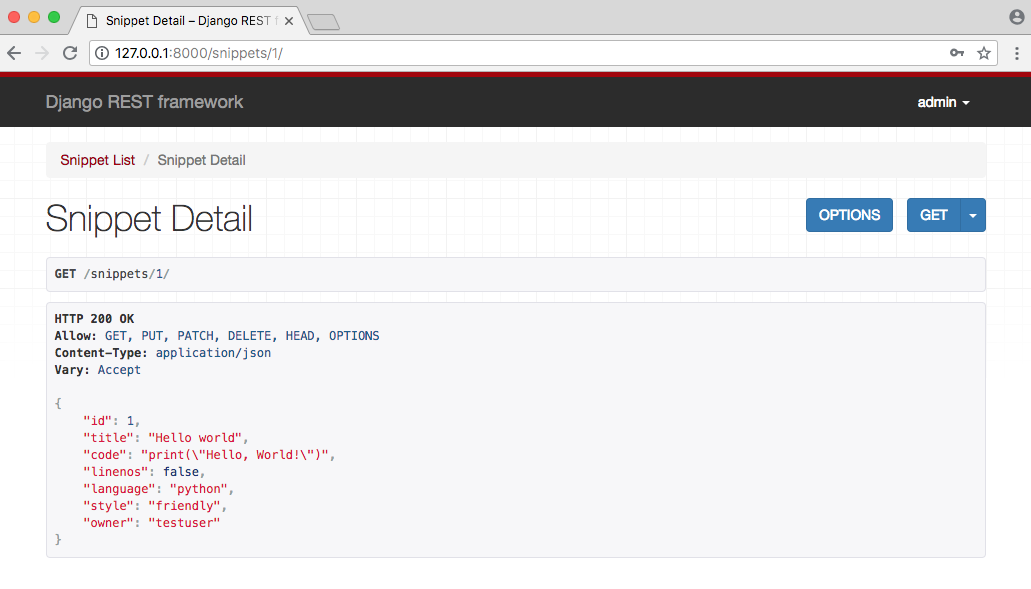
IsOwnerOrReadOnly,) *# new*

**...**

Open the browser again to <http://127.0.0.1:8000/snippets/1/>. Find that the ‘DELETE’ and ‘PUT’ actions appear on the snippet instance endpoint because we’re logged in as testuser, owner of the snippet.



Now log out and log in with admin account. The DELETE and PUT options are **not available**. Good, as expected.



## Root API Endpoint

Currently there are endpoints for snippets and users, but we don’t have a single entry point to our API. To create one, we’ll use a regular function-based view and Django REST Framework’s built-in [@api\_view](http://www.django-rest-framework.org/api-guide/views/#api_view) decorator. In snippets/views.py import api\_view,\ Response, and reverse. Then use @api\_view to set a GET for api\_root.

*# snippets/views.py*

from django.contrib.auth.models import User

from rest\_framework import generics, permissions

from rest\_framework.decorators import api\_view *# new*

from rest\_framework.response import Response *# new*

from rest\_framework.reverse import reverse *# new*

from .models import Snippet

from .permissions import IsOwnerOrReadOnly

from .serializers import SnippetSerializer, UserSerializer

@api\_view(['GET']) *# new*

**def** **api\_root**(request, format**=**None):

**return** Response({

'users': reverse('user-list', request**=**request, format**=**format),

'snippets': reverse('snippet-list', request**=**request, format**=**format)

})

**...**

Next add a URL at the empty string '' for api\_root. And since we’re using reverse, add [named urls](https://docs.djangoproject.com/en/2.0/topics/http/urls/#naming-url-patterns) to each existing view.

*# snippets/urls.py*

from django.urls import path

from rest\_framework.urlpatterns import format\_suffix\_patterns

from snippets import views

urlpatterns **=** [

path('snippets/', views**.**SnippetList**.**as\_view(), name**=**'snippet-list'),

path('snippets/<int:pk>/', views**.**SnippetDetail**.**as\_view(), name**=**'snippet-detail'),

path('users/', views**.**UserList**.**as\_view(), name**=**'user-list'),

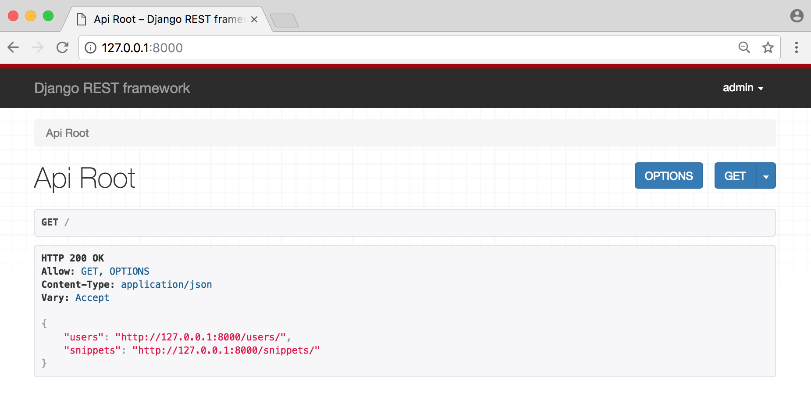
path('users/<int:pk>/', views**.**UserDetail**.**as\_view(), name**=**'user-detail'),

path('', views**.**api\_root),

]

urlpatterns **=** format\_suffix\_patterns(urlpatterns)

Now navigate to <http://127.0.0.1:8000/> to see new API Root page.



It lists both users and snippets as well as their respective API endpoints which can be clicked on.

## Highlighted Snippets Endpoint

The thing that’s still missing from our pastebin API is the code highlighting endpoints. Unlike all our other API endpoints, we don’t want to use JSON, but instead just present an HTML representation. REST framework has two HTML renderers: one for dealing with HTML rendered using templates and one for pre-rendered HTML (which is our case here). Also, there’s no existing generic view that will work so we’ll need to create our own .get() method. In your snippets/views.py import renderers at the top and then create a new class for SnippetHighlight.

*# snippets/views.py*

from django.contrib.auth.models import User

from rest\_framework import generics, permissions, renderers *# new*

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework.reverse import reverse

from .models import Snippet

from .permissions import IsOwnerOrReadOnly

from .serializers import SnippetSerializer, UserSerializer

**class** **SnippetHighlight**(generics**.**GenericAPIView): *# new*

queryset **=** Snippet**.**objects**.**all()

renderer\_classes **=** (renderers**.**StaticHTMLRenderer,)

**def** **get**(self, request, **\***args, **\*\***kwargs):

snippet **=** self**.**get\_object()

**return** Response(snippet**.**highlighted)

**...**

Add the new view to the urls file. Make sure to include the name snippet-highlight!

*# snippets/urls.py*

from django.urls import path

from rest\_framework.urlpatterns import format\_suffix\_patterns

from snippets import views

urlpatterns **=** [

path('snippets/', views**.**SnippetList**.**as\_view(), name**=**'snippet-list'),

path('snippets/<int:pk>/', views**.**SnippetDetail**.**as\_view(), name**=**'snippet-detail'),

path('snippets/<int:pk>/highlight/',

views**.**SnippetHighlight**.**as\_view(), name**=**'snippet-highlight'), *# new*

path('users/', views**.**UserList**.**as\_view(), name**=**'user-list'),

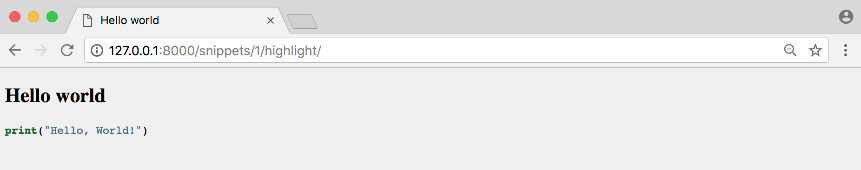
path('users/<int:pk>/', views**.**UserDetail**.**as\_view(), name**=**'user-detail'),

path('', views**.**api\_root),

]

urlpatterns **=** format\_suffix\_patterns(urlpatterns)

We only have one snippet in our database so the highlight will be located at <http://127.0.0.1:8000/snippets/1/highlight/>.



## Hyperlinking our API

One of the more challenging aspects of web API design is dealing with the relationships *between* entities. We could use primary key, hyperlinks, slugs, strings, nesting, or a custom representation.

REST framework supports all of these styles but here we’ll use a hyperlinked style between entities. In order to do so, we’ll modify our serializers to extend HyperlinkedModelSerializer instead of the existing ModelSerializer. The HyperlinkedModelSerializer has the following differences from ModelSerializer:

* It does not include the id field by default.
* It includes a url field, using HyperlinkedIdentityField.
* Relationships use HyperlinkedRelatedField, instead of PrimaryKeyRelatedField.

Let’s rewrite our existing serializers to use hyperlinks.

*# snippets/serializers.py*

from django.contrib.auth.models import User

from rest\_framework import serializers

from snippets.models import Snippet, LANGUAGE\_CHOICES, STYLE\_CHOICES

**class** **SnippetSerializer**(serializers**.**HyperlinkedModelSerializer): *# new*

owner **=** serializers**.**ReadOnlyField(source**=**'owner.username')

highlight **=** serializers**.**HyperlinkedIdentityField( *# new*

view\_name**=**'snippet-highlight', format**=**'html')

**class** **Meta**:

model **=** Snippet

fields **=** ('url', 'id', 'highlight', 'title', 'code', 'linenos',

'language', 'style', 'owner',) *# new*

**class** **UserSerializer**(serializers**.**HyperlinkedModelSerializer): *# new*

snippets **=** serializers**.**HyperlinkedRelatedField( *# new*

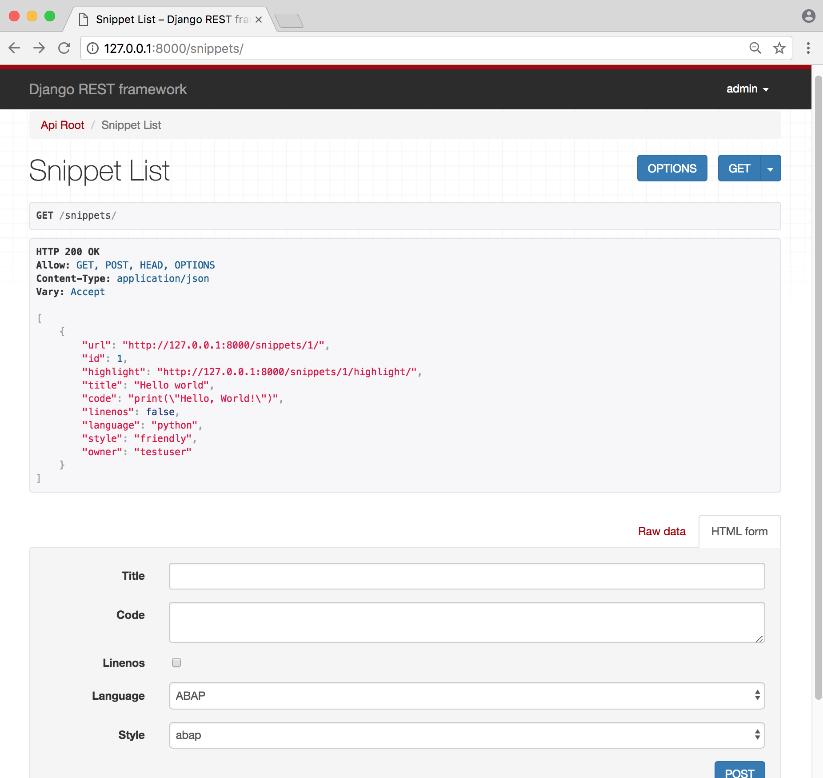
many**=**True, view\_name**=**'snippet-detail', read\_only**=**True)

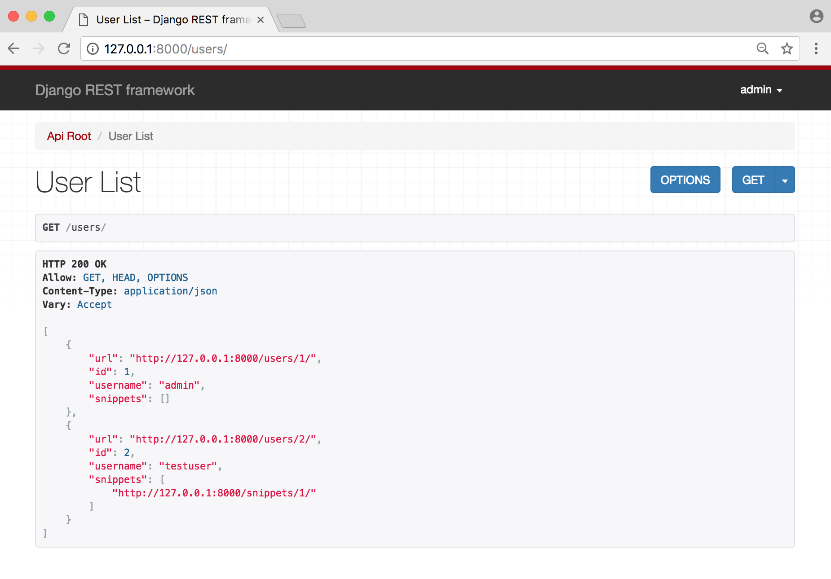
**class** **Meta**:

model **=** User

fields **=** ('url', 'id', 'username', 'snippets') *# new*

Aside from swapping in HyperlinkedModelSerializer there is a new highlight field for snippets that points to snippet-highlight url pattern, instead of snippet-detail url pattern. For the fields we add url to both and highlight to snippet serializer.





## Pagination

Currently we only have the one code snippet but as others are added it makes sense to limit the number of snippets displayed per API endpoint. Let’s paginate the results so that API clients can step through each of the individual pages. REST Framework ships with a number of [default settings](http://www.django-rest-framework.org/api-guide/settings/) which can be easily customized. We’ll set a DEFAULT\_PAGINATION\_CLASS and PAGE\_SIZE to 10 although we could easily customize things further as desired.

At the bottom of the tutorial/settings.py file add the following:

*# tutorial/settings.py*

REST\_FRAMEWORK **=** {

'DEFAULT\_PAGINATION\_CLASS':

'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10

}

At this point it’s possible to click around the entire API just via links. Success!

## Viewsets and Routers

[Section 6](http://www.django-rest-framework.org/tutorial/6-viewsets-and-routers/) of the official tutorial has us switch over from views and URLs to viewsets and routers. This is an optional choice that is, in my opinion, better suited to larger API projects and for developers *already* comfortable with REST framework. Since neither applies here we will not update our code. The resulting API endpoints will still be *exactly the same*!

## Schemas

A **schema** is a machine-readable document that describes the available API endpoints, their URLS, and what operations they support. Schemas can be a useful tool for auto-generated documentation, and can also be used to drive dynamic client libraries that can interact with the API. In order to provide schema support REST framework, use [Core API](http://www.coreapi.org/). We’ll also need to install [pyyaml](https://pyyaml.org/) so that we can render the schema into the commonly used YAML-based OpenAPI format. First stop the local server Control+c and install both packages.

(env) $ pipenv install coreapi pyyaml

Then add a URL route for it. Import get\_schema\_view at the top, create a schema\_view, and then a route at schema/.

*# snippets/urls.py*

from django.urls import path

from rest\_framework.urlpatterns import format\_suffix\_patterns

from rest\_framework.schemas import get\_schema\_view *# new*

from snippets import views

schema\_view **=** get\_schema\_view(title**=**'Pastebin API') *# new*

urlpatterns **=** [

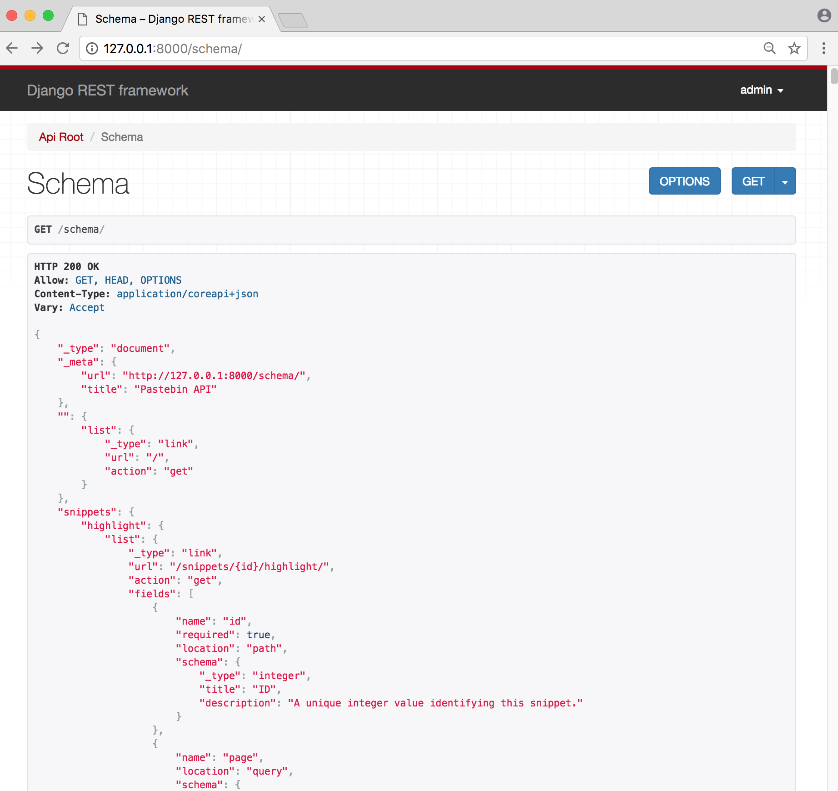
path('schema/', schema\_view), *# new*

**...**

]

urlpatterns **=** format\_suffix\_patterns(urlpatterns)

Now start up the local server again with python manage.py runserver and navigate to <http://127.0.0.1:8000/schema/>.



Voila! An auto-generated schema of our API.

## Command line client

Now that our API is exposing a schema endpoint, we can use a dynamic client library to interact with the API. To demonstrate this, let’s use the Core API command line client. We need two command line consoles at this point. Our existing one should still be running the local server. But we need a second, new console to execute our commands. So open a **second command line console**, navigate to our current directory, and then make sure to activate the virtual environment with pipenv shell so that you see (env) or the equivalent on the command line. Now install coreapi-cli in this new, second console.

(env) $ pipenv install coreapi-cli

To make sure it is working, type coreapi and hit Return. You should see the following.

(env) $ coreapi

Usage: coreapi [OPTIONS] COMMAND [ARGS]...

Command line client for interacting with CoreAPI services.

Visit [http://www.coreapi.org](http://www.coreapi.org/) for more information.

Options:

--version Display the package version number. --help Show this message and exit.

Commands: action Interact with the active document.

bookmarks Add, remove and show bookmarks.

clear Clear the active document and other state. codecs Manage the installed codecs.

credentials Configure request credentials.

describe Display description for link at given PATH. dump Dump a document to console.

get Fetch a document from the given URL.

headers Configure custom request headers. history Navigate the browser history.

load Load a document from disk.

reload Reload the current document. show Display the current document.

Ok good. Now load the API schema by performing a get request on our schema/endpoint.

(env) $ coreapi get <http://127.0.0.1:8000/schema/>

<Pastebin API "<http://127.0.0.1:8000/schema/>">

: { list()

}

snippets: { highlight: {

list(id, [page])

} list([page])

read(id)

read\_0(format, id) read\_1(format, id)

}

users: { list([page])

read(id)

read\_0(format, id) }

read(format)

read\_0(format) read\_1(format)

We haven’t authenticated yet, so right now we’re only able to see the read only endpoints, in line with how we’ve set up the permissions on the API.

Let’s try the existing snippets:

(env) $ coreapi action snippets list

{

"count": 1, "next": null,

"previous": null,

"results": [ {

"url": "<http://127.0.0.1:8000/snippets/1/>",

"id": 1, "highlight": "<http://127.0.0.1:8000/snippets/1/highlight/>",

"title": "Hello world",

"code": "print(\"Hello, World!\")", "linenos": false,

"language": "python",

"style": "friendly", "owner": "testuser"

}

]

}

Now let’s try a highlight endpoint that returns HTML.

(env) $ coreapi get <http://127.0.0.1:8000/snippets/1/highlight/>

**<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "**[**http://www.w3.org/TR/html4/strict.dtd**](http://www.w3.org/TR/html4/strict.dtd)**">**

<html><head>

<title>Hello world</title>

<meta http-equiv="content-type" content="text/html; charset=None"> <style type="text/css">td.linenos { background-color: #f0f0f0; padding-right: 10px; }

...

The result is the pre-formatted HTML of the page.

## Authenticating our client

If we want to be able to create, edit and delete snippets, we’ll need to authenticate as a valid user. REST Framework ships with [multiple authentication options](http://www.django-rest-framework.org/api-guide/authentication/) but in this case we’ll just use basic auth which is the default setting.

Make sure to replace the <username> and <password> below with your actual username and password. Mine are admin and testpass123.

(env) $ coreapi credentials add 127.0.0.1 admin:testpass123 --auth basic

Added credentials

127.0.0.1 "Basic YWRtaW46d2lsbGlhbTE="

Now that we are authenticated we should be able to interact with all API endpoints.

For example to create a new snippet

(env) $ coreapi action snippets create --param title="Example" --param co

de="print('hello, world')"

{

"url": "<http://127.0.0.1:8000/snippets/2/>",

"id": 2,

"highlight": "<http://127.0.0.1:8000/snippets/2/highlight/>",

"title": "Example",

"code": "print('hello, world')",

"linenos": false,

"language": "python",

"style": "friendly",

"owner": "admin"

}

And to then delete that snippet:

(env) $ coreapi action snippets delete --param id=2

Note that there is currently an issue with coreapi so creating a snippet on the command-line [may not work until the bug is fixed](https://stackoverflow.com/questions/44201598/coreapi-only-lists-list-and-read-method-even-when-user-is-logged). However you can do it with the browsable API or a third-party client library like [Postman](https://www.getpostman.com/).

## Conclusion

With an incredibly small amount of code, we’ve now got a complete pastebin Web API, which is fully web browsable, includes a schema-driven client library, and comes complete with authentication, per-object permissions, and multiple renderer formats.

We’ve walked through each step of the design process, and seen how if we need to customize anything we can gradually work our way down to simply using regular Django views.

You can review the final tutorial code [on Github](https://github.com/wsvincent/rest-framework-tutorial).

# Design philosophies

This document explains some of the fundamental philosophies Django’s developers have used in creating the framework. Its goal is to explain the past and guide the future.

## Overall

### Loose coupling

A fundamental goal of Django’s stack is [loose coupling and tight cohesion](http://wiki.c2.com/?CouplingAndCohesion). The various layers of the framework shouldn’t “know” about each other unless absolutely necessary. For example, the template system knows nothing about Web requests, the database layer knows nothing about data display and the view system doesn’t care which template system a programmer uses. Although Django comes with a full stack for convenience, the pieces of the stack are independent of another wherever possible.

### Less code

Django apps should use as little code as possible; they should lack boilerplate. Django should take full advantage of Python’s dynamic capabilities, such as introspection.

### Quick development

The point of a Web framework in the 21st century is to make the tedious aspects of Web development fast. Django should allow for incredibly quick Web development.

### Don’t repeat yourself (DRY)

Every distinct concept and/or piece of data should live in one, and only one, place. Redundancy is bad. Normalization is good.

The framework, within reason, should deduce as much as possible from as little as possible.

See also

The [discussion of DRY on the Portland Pattern Repository](http://wiki.c2.com/?DontRepeatYourself)

### Explicit is better than implicit

This is a core Python principle listed in [**PEP 20**](https://www.python.org/dev/peps/pep-0020), and it means Django shouldn’t do too much “magic.” Magic shouldn’t happen unless there’s a really good reason for it. Magic is worth using only if it creates a huge convenience unattainable in other ways, and it isn’t implemented in a way that confuses developers who are trying to learn how to use the feature.

### Consistency

The framework should be consistent at all levels. Consistency applies to everything from low-level (the Python coding style used) to high-level (the “experience” of using Django).

## Models

### Explicit is better than implicit

Fields shouldn’t assume certain behaviors based solely on the name of the field. This requires too much knowledge of the system and is prone to errors. Instead, behaviors should be based on keyword arguments and, in some cases, on the type of the field.

### Include all relevant domain logic

Models should encapsulate every aspect of an “object,” following Martin Fowler’s [Active Record](https://www.martinfowler.com/eaaCatalog/activeRecord.html) design pattern.

This is why both the data represented by a model and information about it (its human-readable name, options like default ordering, etc.) are defined in the model class; all the information needed to understand a given model should be stored *in* the model.

## Database API

The core goals of the database API are:

### SQL efficiency

It should execute SQL statements as few times as possible, and it should optimize statements internally.

This is why developers need to call save() explicitly, rather than the framework saving things behind the scenes silently.

This is also why the select\_related() QuerySet method exists. It’s an optional performance booster for the common case of selecting “every related object.”

### Terse, powerful syntax

The database API should allow rich, expressive statements in as little syntax as possible. It should not rely on importing other modules or helper objects.

Joins should be performed automatically, behind the scenes, when necessary.

Every object should be able to access every related object, systemwide. This access should work both ways.

### Option to drop into raw SQL easily, when needed

The database API should realize it’s a shortcut but not necessarily an end-all-be-all. The framework should make it easy to write custom SQL – entire statements, or just custom WHERE clauses as custom parameters to API calls.

## URL design

### Loose coupling

URLs in a Django app should not be coupled to the underlying Python code. Tying URLs to Python function names is a Bad And Ugly Thing.

Along these lines, the Django URL system should allow URLs for the same app to be different in different contexts. For example, one site may put stories at /stories/, while another may use /news/.

### Infinite flexibility

URLs should be as flexible as possible. Any conceivable URL design should be allowed.

### Encourage best practices

The framework should make it just as easy (or even easier) for a developer to design pretty URLs than ugly ones.

File extensions in Web-page URLs should be avoided.

Vignette-style commas in URLs deserve severe punishment.

### Definitive URLs

Technically, foo.com/bar and foo.com/bar/ are two different URLs, and search-engine robots (and some Web traffic-analyzing tools) would treat them as separate pages. Django should make an effort to “normalize” URLs so that search-engine robots don’t get confused.

This is the reasoning behind the [APPEND\_SLASH](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-APPEND_SLASH) setting.

## Template system

### Separate logic from presentation

We see a template system as a tool that controls presentation and presentation-related logic – and that’s it. The template system shouldn’t support functionality that goes beyond this basic goal.

### Discourage redundancy

The majority of dynamic websites use some sort of common sitewide design – a common header, footer, navigation bar, etc. The Django template system should make it easy to store those elements in a single place, eliminating duplicate code.

This is the philosophy behind [template inheritance](https://docs.djangoproject.com/en/dev/ref/templates/language/#template-inheritance).

### Be decoupled from HTML

The template system shouldn’t be designed so that it only outputs HTML. It should be equally good at generating other text-based formats, or just plain text.

### XML should not be used for template languages

Using an XML engine to parse templates introduces a whole new world of human error in editing templates – and incurs an unacceptable level of overhead in template processing.

### Assume designer competence

The template system shouldn’t be designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe of a limitation and wouldn’t allow the syntax to be as nice as it is. Django expects template authors are comfortable editing HTML directly.

### Treat whitespace obviously

The template system shouldn’t do magic things with whitespace. If a template includes whitespace, the system should treat the whitespace as it treats text – just display it. Any whitespace that’s not in a template tag should be displayed.

### Don’t invent a programming language

The goal is not to invent a programming language. The goal is to offer just enough programming-esque functionality, such as branching and looping, that is essential for making presentation-related decisions. The [Django Template Language (DTL)](https://docs.djangoproject.com/en/dev/topics/templates/#template-language-intro) aims to avoid advanced logic.

The Django template system recognizes that templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

### Safety and security

The template system, out of the box, should forbid the inclusion of malicious code – such as commands that delete database records.

This is another reason the template system doesn’t allow arbitrary Python code.

### Extensibility

The template system should recognize that advanced template authors may want to extend its technology.

This is the philosophy behind custom template tags and filters.

## Views

### Simplicity

Writing a view should be as simple as writing a Python function. Developers shouldn’t have to instantiate a class when a function will do.

### Use request objects

Views should have access to a request object – an object that stores metadata about the current request. The object should be passed directly to a view function, rather than the view function having to access the request data from a global variable. This makes it light, clean and easy to test views by passing in “fake” request objects.

### Loose coupling

A view shouldn’t care about which template system the developer uses – or even whether a template system is used at all.

### Differentiate between GET and POST

GET and POST are distinct; developers should explicitly use one or the other. The framework should make it easy to distinguish between GET and POST data.

## Cache Framework

The core goals of Django’s [cache framework](https://docs.djangoproject.com/en/dev/topics/cache/) are:

### Less code

A cache should be as fast as possible. Hence, all framework code surrounding the cache backend should be kept to the absolute minimum, especially for get() operations.

### Consistency

The cache API should provide a consistent interface across the different cache backends.

### Extensibility

The cache API should be extensible at the application level based on the developer’s needs (for example, see [Cache key transformation](https://docs.djangoproject.com/en/dev/topics/cache/#cache-key-transformation)).

# Django Object-Relational Mapper

The [Django](https://www.fullstackpython.com/django.html) [web framework](https://www.fullstackpython.com/web-frameworks.html) includes a default [object-relational mapping layer (ORM)](https://www.fullstackpython.com/object-relational-mappers-orms.html) that can be used to interact with application [data](https://www.fullstackpython.com/data.html) from various [relational databases](https://www.fullstackpython.com/databases.html) such as [SQLite](https://www.fullstackpython.com/sqlite.html), [PostgreSQL](https://www.fullstackpython.com/postgresql.html) and [MySQL](https://www.fullstackpython.com/mysql.html).

The Django ORM is an implementation of the [object-relational mapping (ORM)](https://www.fullstackpython.com/object-relational-mappers-orms.html) concept. Learn more in the [data](https://www.fullstackpython.com/data.html) chapter or view [all topics](https://www.fullstackpython.com/table-of-contents.html).

## Django ORM resources

The Django ORM has evolved over the past dozen years since it was created make sure to not only read up on the latest tutorials but also learn about newer optimizations, such as [prefetch\_related](https://docs.djangoproject.com/en/1.11/ref/models/querysets/#prefetch-related) and [select\_related](https://docs.djangoproject.com/en/1.11/ref/models/querysets/#select-related), that have been added throughout the project's history.

* [Django models, encapsulation and data integrity](http://www.dabapps.com/blog/django-models-and-encapsulation/) is a detailed article by Tom Christie on encapsulating Django models for data integrity.
* [Django Debug Toolbar](http://django-debug-toolbar.readthedocs.org/en/1.8/) is a powerful Django ORM database query inspection tool. Highly recommended during development to ensure you're writing reasonable query code. [Django Silk](http://mtford.co.uk/blog/2/) is another inspection tool and has capabilities to do more than just SQL inspection.
* [Making a specific Django app faster](http://reinout.vanrees.org/weblog/2014/05/06/making-faster.html) is a Django performance blog post with some tips on measuring performance and optimizing based on the measured results.
* [Why I Hate the Django ORM](https://speakerdeck.com/alex/why-i-hate-the-django-orm) is Alex Gaynor's overview of the bad designs decisions, some of which he made, while building the Django ORM.
* [Going Beyond Django ORM with Postgres](https://speakerdeck.com/craigkerstiens/going-beyond-django-orm-with-postgres) is specific to using PostgreSQL with Django.
* [How to view Django ORM SQL queries](https://mrcoles.com/how-view-django-orm-sql-queries/) along with [django-sql-explorer](https://github.com/groveco/django-sql-explorer) allow you to better understand the SQL code that is generated from the Django ORM.
* [Migrating a Django app from MySQL to PostgreSQL](http://www.calazan.com/migrating-django-app-from-mysql-to-postgresql/) is a quick look at how to move from MySQL to PostgreSQL. However, my guess is that any Django app that's been running for awhile on one [relational database](https://www.fullstackpython.com/databases.html) will require a lot more work to port over to another backend even with the power of the ORM.
* [Adding basic search to your Django site](https://www.calazan.com/adding-basic-search-to-your-django-site/) shows how to write generic queries that'll allow you to provide site search via the Django ORM without relying on another tool like ElasticSearch. This is great for small sites before you scale them up with a more robust search engine.
* The [Django ORM Cookbook](https://books.agiliq.com/projects/django-orm-cookbook/en/latest/) provides code recipes for various ways to use the Django ORM to insert and query data.
* [How to use Django's Proxy Models](http://benlopatin.com/using-django-proxy-models/) is a solid post on a Django ORM concept that doesn't frequently get a lot of love or explanation.
* [Tightening Django Admin Logins](http://tech.marksblogg.com/django-admin-logins.html) shows you how to log authentication failures, create an IP addresses white list and combine fail2ban with the authentication failures list.
* [How to Turn Django Admin Into a Lightweight Dashboard](https://hakibenita.com/how-to-turn-django-admin-into-a-lightweight-dashboard) and [How to Use Grouping Sets in Django](https://hakibenita.com/how-to-use-grouping-sets-in-django) are two great posts on how to add custom features to the Django Admin as well as optimize with more advanced SQL when the first attempt at the queries get slow due to larger amounts of data.
* [Sorting querysets with NULLs in Django](https://www.isotoma.com/blog/2015/11/23/sorting-querysets-with-nulls-in-django/) shows what to do if you're struggling with the common issue of sorting columns that contain NULL values.
* [Best Practices working with Django models in Python](http://steelkiwi.com/blog/best-practices-working-django-models-python/) has a ton of great advice on proper model naming conventions, quirks to avoid with ForeignKey field relationships, handling IDs and many other edge cases that come up when frequently working with Django's ORM.
* [Merging Django ORM with SQLAlchemy for Easier Data Analysis](https://djangostars.com/blog/merging-django-orm-with-sqlalchemy-for-easier-data-analysis/) provides rationale for using the [SQLAlchemy](https://www.fullstackpython.com/sqlalchemy.html) ORM instead of Django's default ORM in some situations.
* [Working with huge data sets in Django](https://blog.labdigital.nl/working-with-huge-data-sets-in-django-169453bca049) explains how to slice the data you retrieve by query into pages and then use prefetch\_related on a subset of the data rather than your whole data set.
* [Solving performance problems in the Django ORM](https://medium.com/@hansonkd/performance-problems-in-the-django-orm-1f62b3d04785) gives a slew of great code snippets to use with django.db.connection so you can discover issues such as unexpected extra queries and problematic key relationships.
* [Full-text search in Django with PostgreSQL](https://www.paulox.net/2017/12/22/full-text-search-in-django-with-postgresql/) is a very detailed example that shows how to work specifically with a [PostgreSQL](https://www.fullstackpython.com/postgresql.html) backend.
* [Django Anti-Patterns: Signals](https://lincolnloop.com/blog/django-anti-patterns-signals/) explains why you should avoid using Django ORM's [signals](https://docs.djangoproject.com/en/dev/topics/signals/) feature in your applications if you want to make them easier to maintain.
* [Django ORM optimization story on selecting the least possible](https://www.peterbe.com/plog/django-orm-optimization-story-on-selecting-the-least-possible) goes through one developer's Django ORM code refactoring to optimize the performance and results of a single query.
* [Fixing your Django async job - database integration](https://spapas.github.io/2019/02/25/django-fix-async-db/) is a great article on how to properly integrate the [RQ task queue](https://www.fullstackpython.com/redis-queue-rq.html) with a Django backend.

## Django migrations resources

[Django migrations](https://docs.djangoproject.com/en/dev/topics/migrations/) were added in [version 1.7](https://docs.djangoproject.com/en/dev/releases/1.7/). Django projects prior to 1.7 used the [South project](https://south.readthedocs.io/en/latest/), which is now deprecated and merged into Django. Migrations can be tricky to wrap your head around as you're getting started with the overall framework but the following resources should get you past the initial hurdles.

* [Django Migrations - a Primer](https://realpython.com/blog/python/django-migrations-a-primer/) takes you through the new migrations system integrated in the Django core as of Django 1.7, looking specifically at a solid workflow that you can use for creating and applying migrations.
* [Django 1.7: Database Migrations Done Right](https://markusholtermann.eu/2014/09/django-17-database-migrations-done-right/) explains why South was not directly integrated into Django, how migrations are built and shows how backwards migrations work.
* [Executing custom SQL in Django migrations](https://www.endpoint.com/blog/2016/09/17/executing-custom-sql-in-django-migration) examines how you can hook in straight SQL that will run during a Django migration.
* [Squashing and optimizing migrations in Django](http://www.rkblog.rk.edu.pl/w/p/squashing-and-optimizing-migrations-django/) shows a simple example with code for how to use the migrations integrated into Django 1.7.
* [Supporting both Django 1.7 and South](http://treyhunner.com/2014/03/migrating-to-django-1-dot-7/) explains the difficulty of supporting Django 1.7 and maintaining South migrations for Django 1.6 then goes into how it can be done.
* [Writing unit tests for Django migrations](https://www.caktusgroup.com/blog/2016/02/02/writing-unit-tests-django-migrations/) contains a ton of awesome code examples for testing your migrations to ensure data migrations work well throughout the lifecycle of your Django project.
* [Strategies for reducing memory usage in Django migrations](https://www.azavea.com/blog/2017/02/23/strategies-reducing-memory-usage-django-migrations/) shows the large memory usage problem that often occurs with Django migrations at scale and what you can do to mitigate the issue.
* [How to Create Django Data Migrations](https://simpleisbetterthancomplex.com/tutorial/2017/09/26/how-to-create-django-data-migrations.html) has a straightforward blog ORM modeling example to show how to perform data migration.
* [Keeping data integrity with Django migrations](https://cheesecakelabs.com/blog/keeping-data-integrity-django-migrations/) shows two table modification scenarios, one where a column needs to be added to an existing table, and another where a Many-to-Many field needs to be converted to a standard ForeignKey column while retaining all of the data.

## Shortcomings in Django ORM: look at Peewee

December 15, 2012 14:27 / [django](http://charlesleifer.com/blog/tags/django/) [peewee](http://charlesleifer.com/blog/tags/peewee/) [python](http://charlesleifer.com/blog/tags/python/) / [14 comments](http://charlesleifer.com/blog/shortcomings-in-the-django-orm-and-a-look-at-peewee-a-lightweight-alternative/#comments)

In this post I'd like to talk about some of the shortcomings of the Django ORM, the ways [peewee](http://docs.peewee-orm.com) approaches things differently, and how this resulted in peewee having an API that is both more consistent *and* more expressive.

[Alex Gaynor](http://alexgaynor.net/), one of the more outspoken core developers on the Django project, gave a great talk at ChiPY titled ["Why I Hate the Django ORM"](http://www.youtube.com/watch?v=GxL9MnWlCwo) ([slides](https://speakerdeck.com/alex/why-i-hate-the-django-orm)). I think he did a great job identifying what I agree are the two biggest issues with Django's ORM:

* inconsistent API
* lack of composability

# The Django ORM inconsistent API

Django [wants to expose consistent APIs](https://docs.djangoproject.com/en/dev/misc/design-philosophies/#consistency)

The framework should be consistent at all levels. Consistency applies to everything from low-level (the Python coding style used) to high-level (the “experience” of using Django).

On the whole I think Django does a good job with this - the big exception being the ORM. Alex gives 4 examples, which should be familiar to Django developers and which are indicative of the underlying issues:

* filter(field\_name=value)
* Q(field\_name=value)
* F('field\_name')
* Aggregate('field\_name')

**filter()** is the most basic method and is the most common. It is used to express a SQL **Where** clause and takes as parameters a list of keyword arguments mapping field names to values, *and* it can take a list of **Q** objects. **Q** objects also take a keyword argument and are used to allow "combining" one or more expressions into a query tree (this is how you express a logical "OR").

This is the first inconsistency -- there are two methods of expressing the same query, one is just more "specialized":

>>> sq1 = str(Blog.objects.filter(author='charlie').query)

>>> sq2 = str(Blog.objects.filter(Q(author='charlie')).query)

>>> sq1 == sq2

True

What happens if you want to reference the value of another column in your call to filter? Maybe there is an Employee model that stores their actual salary and their desired salary...for this you use the **F** class:

>>> Employee.objects.filter(desired\_salary\_\_lt =F('current\_salary'))

Here we see two ways of identifying fields -- as keyword arguments passed to filter and Q, and as strings when passed to F.

Another common operation is to take the aggregate of the values in a particular common, like the SUM or COUNT, or GROUP BY a particular column. There are a [ton of questions](http://stackoverflow.com/search?q=django+group+by+count&submit=search) on stackoverflow asking over how to do this. The answer involves using three new APIs:

* values() - takes a list of string field names
* annotate() - takes an "aggregate" function
* Count(), Sum(), etc - take a string field name

The fact that all these specialized functions are needed to express a fairly common query, and the fact that they all require a specialized API, is a sign of a lurking design problem. For anything more than a simple **Where** clause, Django quickly bogs down in its own APIs.

## Lack of composability

Django also [wants to expose a powerful, expressive querying API](https://docs.djangoproject.com/en/dev/misc/design-philosophies/#terse-powerful-syntax)

The database API should allow rich, expressive statements in as little syntax as possible. It should not rely on importing other modules or helper objects.

Django falls very short on the first part. Rich, expressive statements are simply not possible in the ORM *unless a special API was designed for it* (like Q and F objects). To create a rich API, it is necessary to allow composability -- small pieces can be composed to create larger, more complex pieces.

Alex gives a good example:

SELECT event\_type, SUM(end\_time - start\_time) AS total\_time

FROM event

GROUP BY event\_type

Which he expresses using Django:

Event.objects.values('event\_type').annotate(

total\_time=Sum(F('end\_time') - F('start\_time'))

)

This actually does not work, Django chokes because the **Sum** function does not know how to handle an expression. Essentially anything beyond the very straightforward use-cases the existing APIs were *designed for* will probably not work or will have been addressed by some special-case logic.

Django's ORM falls short on the second part of the design goal as well, requiring users to import special functions for aggregation and use odd one-letter classes ("Q" and "F") to express certain types of queries.

## Learning from Django

When I first wrote peewee I based a lot of my APIs on those I was familiar with. Coming from Django this meant "kwargs"-style querying and double-underscore lookups. As peewee grew and I added features, I mimicked django and added APIs for expressing logical "OR" (Q objects) and column-to-column comparison (F objects). Before long the code was a mess and people were submitting issues when they tried to express a query I **hadn't planned for**. I think that this is a parallel to how the Django ORM has grown up.

I decided to rewrite. It was the best decision I could have made and I learned a ton in the process. I don't have near the number of users, so this was an acceptable path for my project. Peewee is also *only* an ORM, so the argument for making it a *better* ORM outweighs some other concerns.

While I rewrote I decided to focus entirely on the mechanics of expressing rich queries. To do this I took a look at all the atoms that comprise a SQL query, how they interact, and how they are composed. I identified a few things:

* Clauses
* Columns
* Scalars and Parameters (e.g. LIMIT 100 or foo = "Bar")
* Functions

Clauses are things like **SELECT** and **FROM** which denote particular parts of the query -- these I decided to expose as methods on a class. Columns, or field instances, would be exposed as class attributes. Scalars and parameters were easy, just represent them using python's various types. Finally functions, since they are so diverse and accept varying numbers of parameters, would be exposed dynamically.

This allows us to write the above "events" query:

Event.select(

Event.event\_type,

fn.Sum(Event.end\_time - Event.start\_time).alias('total\_time')

).group\_by(Event.event\_type)

The clauses accept as their arguments one or more columns, scalars, functions, or any combination thereof. These components can be combined using logical OR and AND to create query trees. They also support common operations like addition and subtraction, allowing you to express atomic updates and things like the above example where we subtract one column from another.

## Peewee is more consistent *and* expressive

Peewee shed about a third of its code-base during the rewrite, going from 2400 SLOC to around 1600! It also became much more expressive -- more than once I have written a fairly complex query and been pleasently surprised to see that it **just works**. Rather than losing functionality, I have gained flexibility which in turn produces functionality.

## "Fixing" Django

I think that Django would benefit from a similar rewrite. The ORM is one of the most complicated parts of Django and sees some pretty crazy bugs. I'm sure I'm not the only person who thinks this should happen...the problem is:

*How can the ORM change without breaking backwards compatibility?*

From the beginning Django has been committed to backwards compatibility. This may be one of the biggest contributors to Django's adoption.

I would suggest building out a new API that is similar to peewee's. Since the existing APIs are a subset of the functionality possible in peewee, the existing APIs could be rewritten to use the new APIs and marked for deprecation. When rewriting peewee, I included a backwards-compatible method to allow the "django-style" double-underscore querying that had been possible in the older version.

## What do you think?

What is your take on the Django ORM? If you use something like SQLAlchemy or peewee, how do you feel that it compares to Django's ORM? Are there things Django can do that Peewee or SQLAlchemy *cannot*? One example is that, while peewee supports "[prefetch\_related](https://docs.djangoproject.com/en/dev/ref/models/querysets/#prefetch-related)", it is not quite as powerful as Django's implementation (Full disclosure, I think this feature is kind of gross).

Please feel free to [leave a comment](http://charlesleifer.com/blog/shortcomings-in-the-django-orm-and-a-look-at-peewee-a-lightweight-alternative/#comments) below. I'd also invite you to check out [the peewee documentation](http://docs.peewee-orm.com/) if you'd like to see more examples!

## Links

* ["Why I hate the Django ORM"](http://www.youtube.com/watch?v=GxL9MnWlCwo), talk by Alex Gaynor at ChiPY and [slides](https://speakerdeck.com/alex/why-i-hate-the-django-orm)
* [Django's design philosophy](https://docs.djangoproject.com/en/dev/misc/design-philosophies/)
* [Notes from the peewee rewrite](http://docs.peewee-orm.com/en/latest/peewee/upgrading.html#upgrading)
* [Peewee project page](https://github.com/coleifer/peewee)

# [Documenting your API](https://www.django-rest-framework.org/topics/documenting-your-api/#documenting-your-api)

A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state.

— Roy Fielding, [REST APIs must be hypertext driven](https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)

REST framework provides built-in support for API documentation. There are also a number of great third-party documentation tools available.

## [Built-in API documentation](https://www.django-rest-framework.org/topics/documenting-your-api/#built-in-api-documentation)

The built-in API documentation includes:

* Documentation of API endpoints.
* Automatically generated code samples for each of the available API client libraries.
* Support for API interaction.

### [Installation](https://www.django-rest-framework.org/topics/documenting-your-api/#installation)

The coreapi library is required as a dependency for the API docs. Make sure to install the latest version. The pygments and markdown libraries are optional but recommended.

To install the API documentation, you'll need to include it in your project's URLconf:

from rest\_framework.documentation import include\_docs\_urls

urlpatterns = [

...

url(r'^docs/', include\_docs\_urls(title='My API title'))

]

This will include two different views:

* /docs/ - The documentation page itself.
* /docs/schema.js - A JavaScript resource that exposes the API schema.

**Note**: By default include\_docs\_urls configures the underlying SchemaView to generate *public* schemas. This means that views will not be instantiated with a request instance. i.e. Inside the view self.request will be None.

To be compatible with this behaviour, methods (such as get\_serializer or get\_serializer\_class etc.) which inspect self.request or, particularly, self.request.user may need to be adjusted to handle this case.

You may ensure views are given a request instance by calling include\_docs\_urls with public=False:

from rest\_framework.documentation import include\_docs\_urls

urlpatterns = [

...

# Generate schema with valid `request` instance:

url(r'^docs/', include\_docs\_urls(title='My API title', public=False))

]

### [Documenting your views](https://www.django-rest-framework.org/topics/documenting-your-api/#documenting-your-views)

You can document your views by including docstrings that describe each of the available actions. For example:

class UserList(generics.ListAPIView):

"""

Return a list of all the existing users.

"""

If a view supports multiple methods, you should split your documentation using method: style delimiters.

class UserList(generics.ListCreateAPIView):

"""

get:

Return a list of all the existing users.

post:

Create a new user instance.

"""

When using viewsets, you should use the relevant action names as delimiters.

class UserViewSet(viewsets.ModelViewSet):

"""

retrieve:

Return the given user.

list:

Return a list of all the existing users.

create:

Create a new user instance.

"""

Custom actions on viewsets can also be documented in a similar way using the method names as delimiters or by attaching the documentation to action mapping methods.

class UserViewSet(viewsets.ModelViewset):

...

@action(detail=False, methods=['get', 'post'])

def some\_action(self, request, \*args, \*\*kwargs):

"""

get:

A description of the get method on the custom action.

post:

A description of the post method on the custom action.

"""

@some\_action.mapping.put

def put\_some\_action():

"""

A description of the put method on the custom action.

"""

### [documentation API Reference](https://www.django-rest-framework.org/topics/documenting-your-api/#documentation-api-reference)

The rest\_framework.documentation module provides three helper functions to help configure the interactive API documentation, include\_docs\_urls (usage shown above), get\_docs\_view and get\_schemajs\_view.

include\_docs\_urls employs get\_docs\_view and get\_schemajs\_view to generate the url patterns for the documentation page and JavaScript resource that exposes the API schema respectively. They expose the following options for customisation. (get\_docs\_view and get\_schemajs\_view ultimately call rest\_frameworks.schemas.get\_schema\_view(), see the Schemas docs for more options there.)

#### [include\_docs\_urls](https://www.django-rest-framework.org/topics/documenting-your-api/#include_docs_urls)

* title: Default None. May be used to provide a descriptive title for the schema definition.
* description: Default None. May be used to provide a description for the schema definition.
* schema\_url: Default None. May be used to pass a canonical base URL for the schema.
* public: Default True. Should the schema be considered *public*? If True schema is generated without a request instance being passed to views.
* patterns: Default None. A list of URLs to inspect when generating the schema. If None project's URL conf will be used.
* generator\_class: Default rest\_framework.schemas.SchemaGenerator. May be used to specify a SchemaGenerator subclass to be passed to the SchemaView.
* authentication\_classes: Default api\_settings.DEFAULT\_AUTHENTICATION\_CLASSES. May be used to pass custom authentication classes to the SchemaView.
* permission\_classes: Default api\_settings.DEFAULT\_PERMISSION\_CLASSES May be used to pass custom permission classes to the SchemaView.
* renderer\_classes: Default None. May be used to pass custom renderer classes to the SchemaView.

#### [get\_docs\_view](https://www.django-rest-framework.org/topics/documenting-your-api/#get_docs_view)

* title: Default None. May be used to provide a descriptive title for the schema definition.
* description: Default None. May be used to provide a description for the schema definition.
* schema\_url: Default None. May be used to pass a canonical base URL for the schema.
* public: Default True. If True schema is generated without a request instance being passed to views.
* patterns: Default None. A list of URLs to inspect when generating the schema. If None project's URL conf will be used.
* generator\_class: Default rest\_framework.schemas.SchemaGenerator. May be used to specify a SchemaGenerator subclass to be passed to the SchemaView.
* authentication\_classes: Default api\_settings.DEFAULT\_AUTHENTICATION\_CLASSES. May be used to pass custom authentication classes to the SchemaView.
* permission\_classes: Default api\_settings.DEFAULT\_PERMISSION\_CLASSES. May be used to pass custom permission classes to the SchemaView.
* renderer\_classes: Default None. May be used to pass custom renderer classes to the SchemaView. If None the SchemaView will be configured with DocumentationRenderer and CoreJSONRenderer renderers, corresponding to the (default) html and corejson formats.

#### [get\_schemajs\_view](https://www.django-rest-framework.org/topics/documenting-your-api/#get_schemajs_view)

* title: Default None. May be used to provide a descriptive title for the schema definition.
* description: Default None. May be used to provide a description for the schema definition.
* schema\_url: Default None. May be used to pass a canonical base URL for the schema.
* public: Default True. If True schema is generated without a request instance being passed to views.
* patterns: Default None. A list of URLs to inspect when generating the schema. If None project's URL conf will be used.
* generator\_class: Default rest\_framework.schemas.SchemaGenerator. May be used to specify a SchemaGenerator subclass to be passed to the SchemaView.
* authentication\_classes: Default api\_settings.DEFAULT\_AUTHENTICATION\_CLASSES. May be used to pass custom authentication classes to the SchemaView.
* permission\_classes: Default api\_settings.DEFAULT\_PERMISSION\_CLASSES May be used to pass custom permission classes to the SchemaView.

### [Customising code samples](https://www.django-rest-framework.org/topics/documenting-your-api/#customising-code-samples)

The built-in API documentation includes automatically generated code samples for each of the available API client libraries.

You may customise these samples by subclassing DocumentationRenderer, setting languages to the list of languages you wish to support:

from rest\_framework.renderers import DocumentationRenderer

class CustomRenderer(DocumentationRenderer):

languages = ['ruby', 'go']

For each language you need to provide an intro template, detailing installation instructions and such, plus a generic template for making API requests, that can be filled with individual request details. See the [templates for the bundled languages](https://github.com/encode/django-rest-framework/tree/master/rest_framework/templates/rest_framework/docs/langs) for examples.

### [Third party packages](https://www.django-rest-framework.org/topics/documenting-your-api/#third-party-packages)

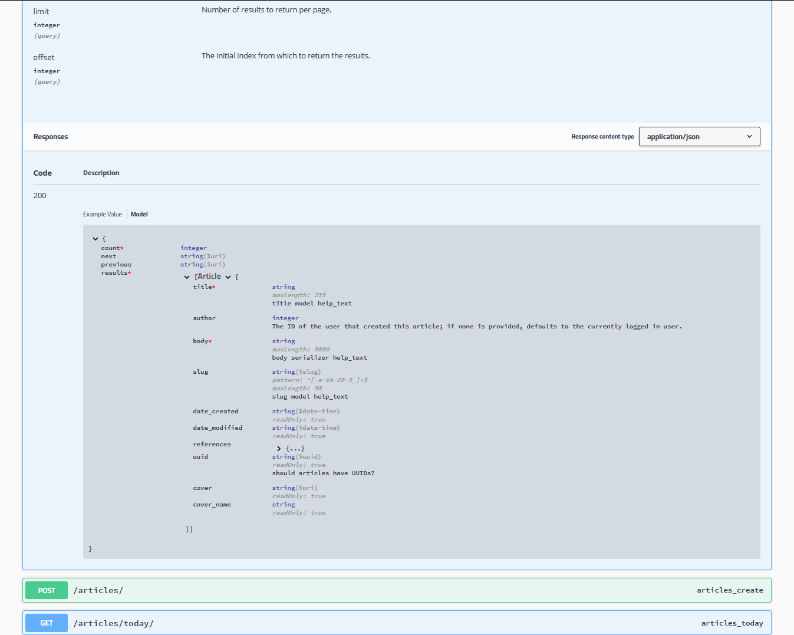
There are a number of mature third-party packages for providing API documentation.

#### [drf-yasg - Yet Another Swagger Generator](https://www.django-rest-framework.org/topics/documenting-your-api/#drf-yasg-yet-another-swagger-generator)

[drf-yasg](https://github.com/axnsan12/drf-yasg/) is a [Swagger](https://swagger.io/) generation tool implemented without using the schema generation provided by Django Rest Framework.

It aims to implement as much of the [OpenAPI](https://openapis.org/) specification as possible - nested schemas, named models, response bodies, enum/pattern/min/max validators, form parameters, etc. - and to generate documents usable with code generation tools like swagger-codegen.

This also translates into a very useful interactive documentation viewer in the form of swagger-ui:

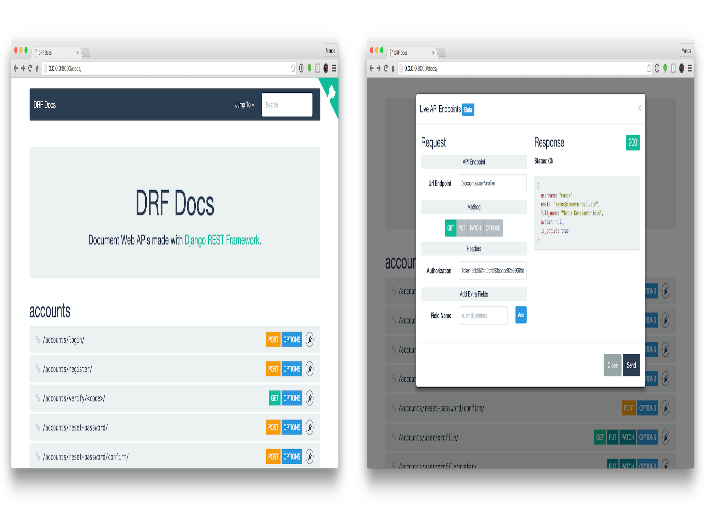


#### [DRF Docs](https://www.django-rest-framework.org/topics/documenting-your-api/#drf-docs)

[DRF Docs](https://github.com/ekonstantinidis/django-rest-framework-docs) allows you to document Web APIs made with Django REST Framework and it is authored by Emmanouil Konstantinidis. It's made to work out of the box and its setup should not take more than a couple of minutes. Complete documentation can be found on the [website](https://www.drfdocs.com/) while there is also a [demo](http://demo.drfdocs.com/) available for people to see what it looks like. **Live API Endpoints** allow you to utilize the endpoints from within the documentation in a neat way.

Features include customizing the template with your branding, settings for hiding the docs depending on the environment and more.

Both this package and Django REST Swagger are fully documented, well supported, and come highly recommended.



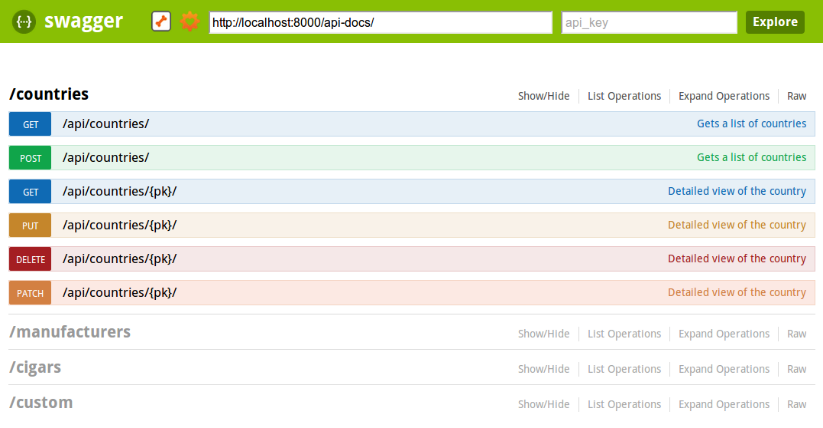
#### [Django REST Swagger](https://www.django-rest-framework.org/topics/documenting-your-api/#django-rest-swagger)

Marc Gibbons' [Django REST Swagger](https://github.com/marcgibbons/django-rest-swagger) integrates REST framework with the [Swagger](https://swagger.io/) API documentation tool. The package produces well presented API documentation, and includes interactive tools for testing API endpoints.

Django REST Swagger supports REST framework versions 2.3 and above.

Mark is also the author of the [REST Framework Docs](https://github.com/marcgibbons/django-rest-framework-docs) package which offers clean, simple autogenerated documentation for your API but is deprecated and has moved to Django REST Swagger.

Both this package and DRF docs are fully documented, well supported, and come highly recommended.



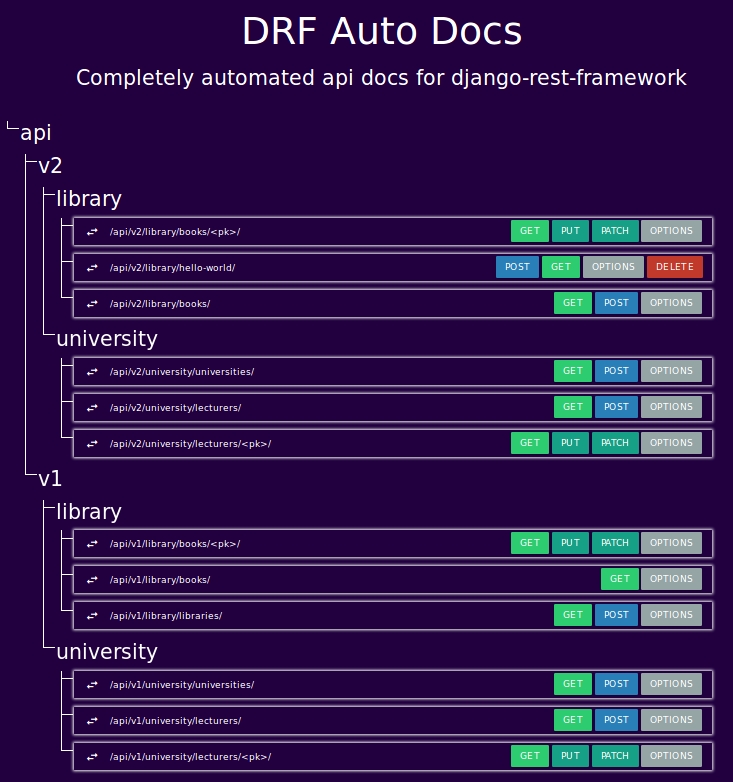
#### [DRF AutoDocs](https://www.django-rest-framework.org/topics/documenting-your-api/#drf-autodocs)

Oleksander Mashianovs' [DRF Auto Docs](https://github.com/iMakedonsky/drf-autodocs) automated api renderer.

Collects almost all the code you written into documentation effortlessly.

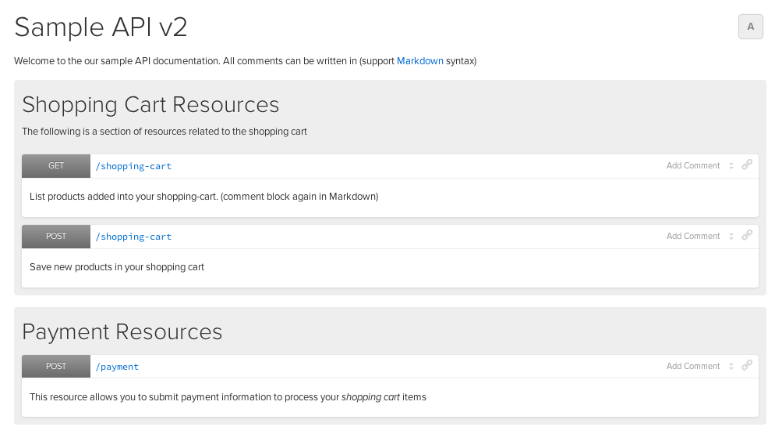
Supports:

* functional view docs
* tree-like structure
* Docstrings:
* markdown
* preserve space & newlines
* formatting with nice syntax
* Fields:
* choices rendering
* help\_text (to specify SerializerMethodField output, etc)
* smart read\_only/required rendering
* Endpoint properties:
* filter\_backends
* authentication\_classes
* permission\_classes
* extra url params(GET params)



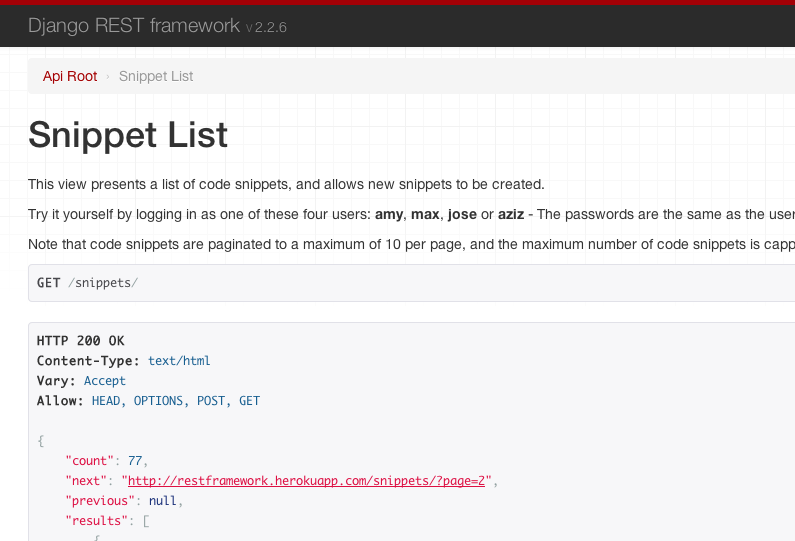
#### [Apiary](https://www.django-rest-framework.org/topics/documenting-your-api/#apiary)

There are various other online tools and services for providing API documentation. One notable service is [Apiary](https://apiary.io/). With Apiary, you describe your API using a simple markdown-like syntax. The generated documentation includes API interaction, a mock server for testing & prototyping, and various other tools.



### [Self describing APIs](https://www.django-rest-framework.org/topics/documenting-your-api/#self-describing-apis)

The browsable API that REST framework provides makes it possible for your API to be entirely self describing. The documentation for each API endpoint can be provided simply by visiting the URL in your browser.



#### [Setting the title](https://www.django-rest-framework.org/topics/documenting-your-api/#setting-the-title)

The title that is used in the browsable API is generated from the view class name or function name. Any trailing View or ViewSet suffix is stripped, and the string is whitespace separated on uppercase/lowercase boundaries or underscores.

For example, the view UserListView, will be named User List when presented in the browsable API.

When working with viewsets, an appropriate suffix is appended to each generated view. For example, the view set UserViewSet will generate views named User List and User Instance.

#### [Setting the description](https://www.django-rest-framework.org/topics/documenting-your-api/#setting-the-description)

The description in the browsable API is generated from the docstring of the view or viewset.

If the python markdown library is installed, then [markdown syntax](https://daringfireball.net/projects/markdown/) may be used in the docstring, and will be converted to HTML in the browsable API. For example:

class AccountListView(views.APIView):

"""

Returns a list of all \*\*active\*\* accounts in the

system. For more details on how accounts are

activated please [see here][ref].

[ref]: http://example.com/activating-accounts

"""

Note that when using viewsets the basic docstring is used for all generated views. To provide descriptions for each view, such as for the the list and retrieve views, use docstring sections as described in [Schemas as documentation: Examples](https://www.django-rest-framework.org/api-guide/schemas/#examples).

#### [The OPTIONS method](https://www.django-rest-framework.org/topics/documenting-your-api/#the-options-method)

REST framework APIs also support programmatically accessible descriptions, using the OPTIONS HTTP method. A view will respond to an OPTIONS request with metadata including the name, description, and the various media types it accepts and responds with.

When using the generic views, any OPTIONS requests will additionally respond with metadata regarding any POST or PUT actions available, describing which fields are on the serializer.

You can modify the response behavior to OPTIONS requests by overriding the options view method and/or by providing a custom Metadata class. For example:

def options(self, request, \*args, \*\*kwargs):

"""

Don't include the view description in OPTIONS responses.

"""

meta = self.metadata\_class()

data = meta.determine\_metadata(request, self)

data.pop('description')

return data

See [the Metadata docs](https://www.django-rest-framework.org/api-guide/metadata/) for more details.

### [The hypermedia approach](https://www.django-rest-framework.org/topics/documenting-your-api/#the-hypermedia-approach)

To be fully RESTful an API should present its available actions as hypermedia controls in the responses that it sends.

In this approach, rather than documenting the available API endpoints up front, the description instead concentrates on the *media types* that are used. The available actions that may be taken on any given URL are not strictly fixed, but are instead made available by the presence of link and form controls in the returned document.

To implement a hypermedia API you'll need to decide on an appropriate media type for the API, and implement a custom renderer and parser for that media type. The [REST, Hypermedia & HATEOAS](https://www.django-rest-framework.org/topics/rest-hypermedia-hateoas/) section of the documentation includes pointers to background reading, as well as links to various hypermedia formats.

# Django vs SQLAlchemy – Which Python ORM is better?

June 8, 2017   
Tomer Shay @ EverSQL

## What’s an ORM?

Before going into the difference between Python’s ORM frameworks (Django and SQLAlchemy), let’s make sure we fully understand the use of ORM frameworks in general.

ORM stands for Object Relational Mapping. Looking at each of these words will explain their use in the real world:

* **Object** – This part represents the objects and programming language where the framework is used, for example Python.
* **Relational** – This part represents the RDBMS database you’re using (Relational Database Manager System). There are numerous popular relational databases out there, but you’re probably using on of the following – MSSQL, MySQL, Oracle Database, PostgreSQL, MariaDB, PerconaDB, TokuDB. What’s common between most relational databases is their relational structures (tables, columns, keys, constraints, etc.).
* **Mapping** – This final part represents the bridge and connection between the two previous parts, the objects and the database tables.

So the conclusion is that the ORM is here to connect between the programming language to the database, in order to simplify the process of creating an application that relies on data.

## Comparing Django vs SQLAlchemy

### Active Record vs Data Mapper

*Django ORM* uses the active record implementation – you’ll see this implementation in most ORMs. Basically what it means is that each row in the database is directly mapped to an object in the code and vice versa. ORM frameworks such as Django won’t require predefining the schema to use the properties in the code. You just use them, as the framework can ‘understand’ the structure by looking at the database schema. Also, you can just save the record to the database, as it’s mapped to a specific row in the table.

*SQLAlchemy* uses the Data Mapper implementation – When using this kind of implementation, there is a separation between the database structure and the objects structure (they are not 1:1 as in the Active Record implementation). In most cases, you’ll have to use another persistence layer to keep interact with the database (for example, to save the object). So you can’t just call the save() method as you can when using the Active Record implementation (which is a con) but on the other hand, you code doesn’t have to know the entire relational structure in the database to function, as there is no direct relationship between the code and the database.

So which of them wins this battle? None. It depends on what you’re trying to accomplish. It’s my believe that if your application is a mostly a CRUD (Create, Read, Update, Delete) application which no hard and complex rules to apply on the relationships between the different data entities, you should use the Active Record implementation (Django). It will allow you to easily and quickly set up an MVP for your product, without any hassle. If you have a lot of “business rules” and restrictions in your applications, you might be better with the Data Mapper model, as it won’t tie you up and force you to think strictly as Active Record does.

### Working with Complex Queries

In some cases, Django and SQLAlchemy can be used together. The main use case I got to see numerous times in the real world is when Django is used for all regular CRUD operations, while SQLAlchemy is used for the more complex queries, usually read-only queries.

For some more information and an example on this aspect, look into [BetterWorks engineering blog](https://engineering.betterworks.com/2015/09/03/sqlalchemy-and-django/) (we have no affiliation with them what so ever, we just liked their blog post :)).

### Primary Key Automatic Generation

Another difference between the two frameworks is that Django can create primary keys automatically for your tables. SQLAlchemy won’t do that for you. You’ll have to manually create them for each table yourself. It’s both a pro and a con – who do you think knows best which primary key will be most suited for your table? It’s up to you to decide, according to your team’s knowledge and experience.

### Autocommit

Django has autocommit on by default, while SQLAlchemy doesn’t. It will impact the way you use the framework (transactions, rollbacks, etc.).

### Supported Databases

Both Django and SQLAlchemy can be used with MySQL, PostgreSQL, Oracle and SQLite. If you’re using MSSQL, you should go for SQLAlchemy, as it’s fully supported by it and you’ll find more information and documentation about it.

### Learning Curve

It’s a common opinion over the web that Django will be a lot easier to learn. Well, it’s probably obvious, as it’s usually used for less complex use cases. So, you should think how much you’re willing to invest in learning the framework, to receive more flexibility in exchange with SQLAlchemy (in case you really need it).

### Community Size

Without any doubt, SQLAlchemy has the largest community among Python ORM frameworks. If community is important to you (and I think it should be), SQL Alchemy should be your choice. It doesn’t mean though that you won’t find any help for other frameworks such as Django. You’ll get bug fixes, answers for questions in StackOverflow and any other help you’ll need, but your chances are just higher with SQLAlchemy.

### Performance

I think it will be irresponsible of me to just write (X is faster than Y) here. It will be very hard to get to that conclusion with ORMs, as they have such a variety of features and capabilities, and they are different in each framework. From my experience, the way you use the framework’s features will have a large impact on the overall performance of your application’s database layer. So my suggestion here is not to choose a framework by its performance, but to learn how to use the framework properly to get the most out of it.

In case you’re running raw SQL queries within an ORM framework, using Jooq or just not using ORM for some of your queries, you can also look into [EverSQL Query Optimizer](http://www.eversql.com) which is probably the easiest way to optimize any query.

## Summary

As in any comparison, I think it’s better to leave the decision making to you, the readers. Each use case is different, and a different technology can be better suited. Look into the differences specified above and let us know which decision you made.

# SQLAlchemy and You

written on Tuesday, July 19, 2011

Without doubt are most new Python web programmers these days chosing the Django framework as their gateway drug to Python web development. As such many people's first experience with a Python ORM (or maybe an ORM altogether) is the Django one. When they are later switching to something else they often find SQLAlchemy unnecessarily complex and hard to use. Why is that the case?

I made a quick poll on Twitter about why people prefer the Django ORM over SQLAlchemy and I got back a few interesting results. First of all that question was obviously asked with the intent to attract answers from people that do prefer Django over SQLAlchemy or at least have some issues with SQLAlchemy that they don't seem to have with Django. Without a doubt there is a large fanbase behind SQLAlchemy, myself included.

SQLAlchemy in general just has a much larger featureset and it's the only ORM for Python which allows you to take full advantage of your database and does not stand in your way. It exposes all features of your underlying database if you want and can be heavily fine tuned.

This article assumes that you have some basic Django knowledge and want to give SQLAlchemy a try. Step by step it walks through the differences and common idioms.

## Design Differences

There are two very important differences between SQLAlchemy and Django. The first one is the less obvious one: SQLAlchemy is a deeply layered system, whereas Django's ORM is basically just one layer which is the ORM you see. In SQLAlchemy you have at the very bottom the engine which with abstracts away connection pools and basic API differences between different databases, on top of that you have the SQL abstraction language, sitting on top of that and the table definitions you have the basic ORM and on top of that you have the declarative ORM which looks very close to the Django ORM. The other more striking difference however is that SQLAlchemy follows the “Unit of Work” pattern whereas Django's ORM follows something that is very close to the “Active Record” pattern.

What's the difference? Django's ORM is basically quite simple. Each time you do any query it generates a SQL expression for you and sends a query to the database. Then it constructs and object for you. That object can be modified and if you call save() on it, it will update the record in the database with the new values of the attributes. This is not at all how SQLAlchemy's ORM component works. In SQLAlchemy you have an object called the “session”. It basically encapsulates a transaction. However it does more. Each object is tracked by primary key in this session. As such each object only exists once by primary key. As such you can safely make a lot of queries and you never have things out of sync. When you commit the session it will send all changes at once to the database in correct order, if you rollback the session nothing happens instead.

## SQLAlchemy's Complexity

SQLAlchemy has to fight with some basic acceptance problems which are caused by the fact that it's framework independent and is not even something you would only use in web applications. This is why projects like [Flask-SQLAlchemy](http://packages.python.org/Flask-SQLAlchemy/) exist to make the integration for you. Many frameworks either provide something that preconfigures SQLAlchemy for you with some sane defaults or have a section in the cookbook to copy/paste code from. This also is the reason why many people find SQLAlchemy's documentation overwhelming. Not only does the documentation guide you through the different levels of SQLAlchemy but also shows all the different ways you can configure SQLAlchemy and the ORM. Django on the other hand does not have many different ways the configure the ORM. It comes preconfigured with the exception of some database related setting such as server name, port and a few other things.

## The Session — The Heart of the ORM

If you chose to use SQLAlchemy's ORM component and not just the engine or SQL abstraction layer you will sooner or later be confronted with the session object. What is the session object? It is the one object that records all the changes on models you do. How does a model know about the session? Let's compare this to Django for a moment. In Django if you have a model it also has an *objects* attribute attached. That attribute points to a manager object which in turn can generate queryset objects and the queryset objects then fire the query and hold the results. How does this queryset object find the current transaction? In Django the answer is that transactions are bound to a thread always. Each thread can have one transaction and the queryset uses. So Django needs to find the transaction in the queryset when the actual query fires and it does that by finding the current thread which owns a transaction. So how does that work in SQLAlchemy? As we have established, objects are always “owned” by a session and keyed by primary key. Each primary key can only exist once. Because that session is quite fundamental and needs to work in many setups this is configurable. But first we need to figure out what the difference between a Django queryset and a SQLAlchemy query object is. They look similar on the surface, but are very different in practice. A Django queryset is created by the manager of a model and also holds the results once the query is fired. In SQLAlchemy the query object can be created in many different ways and unifies the idea of the manager and Django's querysets in itself. When the query object is evaluated (the query is sent to the database) it instead returns a list or the only result object, depending on what methods are used. This means the manager object is entirely unnecessary in SQLAlchemy, if you need custom manager methods you would just subclass the query and attach new methods on there. Since the query object can be joined it makes for quite a nice API. But back to sessions. In Django default of having one transaction per thread makes a lot of sense, but limits the usefulness when you have other means of concurrency somewhat. Also it makes it hard if you want to use the same model against two different databases. This is where the explicit API comes into play which is the one that the documentation uses. So instead of this Django idiom:

MyModel.objects.all()

In default SQLAlchemy you would do this instead:

session.query(MyModel).all()

Here it's explicit what session object should be used and you can have multiple of those side by side obviously. Since many people do not need this and are fine with having one session per thread you can take advantage of the scoped session support in SQLAlchemy. For instance the Flask-SQLAlchemy extension will by default attach a *query* class level attribute to your models which looks at the current thread and it's session object. So each thread will only have one session. Furthermore at the end of an HTTP request in Flask the extension will automatically destroy the session and discarding uncommitted changes. With that, it looks a lot closer to Django:

MyModel.query.all()

You however will still need the session to commit and insert and delete objects from the database. The scoped session automatically provides a proxy that always point to the current active session.

## The Declarative Extension

For a long time SQLAlchemy made you declare table objects first and then separately create the classes and map those together. This has the advantage over just subclassing some magical baseclass that you can map already existing classes to things in the database. The downside always was however that you had to declare multiple objects and the common case was unnecessarily complex.

SQLAlchemy since introduced the declarative base. It's a extension module shipped with SQLAlchemy that provides a function which creates a brand new baseclass (which you can also customize) which does metaclass magic very similar to Django. As such you can directly declare relationships and attributes in the class itself.

There are still some differences though:

* Relationships are not magically created for you, you have to be expicit. The same is true with foreign keys.
* Primary keys are not automatically generated for you for the simple reason that SQLAlchemy supports more than one primary key type. If you want one chosen by default, you can provide a baseclass that implements that.
* The table name has to be set explicitly. You can customize the baseclass to derive the table name from the class name if you like.

To get this baseclass you basically just need this:

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

### Basic Models

A basic Django model looks something like this:

class Person(models.Model):

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

last\_name = models.CharField(max\_length=30)

The equivalent SQLAlchemy model with declarative base:

from sqlalchemy import Column, Integer, String

class Person(Base):

\_\_tablename\_\_ = 'persons'

id = Column(Integer, primary\_key=True)

first\_name = Column(String(30))

last\_name = Column(String(30))

It's a little more to type, but if you want to make this implicit you just need a proper baseclass. Flask-SQLAlchemy for instance sets the lowercase version of the class as default tablename unless overridden.

### Many-to-One Relationships

In Django this is straightforward:

class Manufacturer(models.Model):

name = models.CharField(max\_length=30)

class Car(models.Model):

manufacturer = models.ForeignKey(Manufacturer,

related\_name='cars')

name = models.CharField(max\_length=30)

In SQLAlchemy we have to be a little bit more expressive:

from sqlalchemy import Column, Integer, String, ForeignKey

from sqlalchemy.orm import relationship, backref

class Manufacturer(Base):

\_\_tablename\_\_ = 'manufacturers'

id = Column(Integer, primary\_key=True)

name = Column(String(30))

class Car(models.Model):

\_\_tablename\_\_ = 'cars'

id = Column(Integer, primary\_key=True)

manufacturer\_id = Column(Integer, ForeignKey('manufacturers.id'))

name = Column(String(30))

manufacturer = relationship('Manufacturer', backref=

backref('cars', lazy='dynamic'))

Here we have to model the relationship ourself. First we need to declare the foreign key. It has to have the same type as the primary key of the table we want to point to and additionally the column needs to be given a *ForeignKey* instance with the first argument being the dotted name to the column referenced. Note that this is the table name, not the class name.

The relationship is then declared on *Car* with *relationship*. The first argument is a class or the name of a class we want to have the relationship with. By default it will try to find a valid join condition automatically. If it does not, you can explicitly provide one as a string or real expression:

manufacturer = relationship('Manufacturer',

primaryjoin='Car.manufacturer\_id == Manufacturer.id',

backref=backref('cars', lazy='dynamic'))

The *backref* argument automatically declares the reverse. It will attach a *cars* property on the manufacturer. The lazy='dynamic' tells SQLAlchemy to make the backref lazy and a dynamic loading one. In that case accessing *manufacturer.cars* will be a query object you can further refine instead of directly firing the query and returning a list.

Other lazy settings:

* 'select': if accessed load everything as list with another select statement. This is the default.
* 'joined': uses a join to automatically load that backref with the query of the parent itself.
* 'dynamic': returns a query object instead of firing the query. This can be sliced and further extended.

The lazy settings can also be set on *relationship* and not just backref.

Backref in a nutshell:

'lazy' and 'select'. The first one fires a query when *honda.cars* is accessed, the other one will fetch it when honda is queried:

>>> honda.cars

[<Car 1>, <Car 2>]

And here with 'dynamic':

>>> honda.cars

<AppenderQuery ...>

>>> honda.cars.all()

[<Car 1>, <Car 2>]

### Many-To-Many

Many to many relationships in Django are easy cake because everything is done for you:

class Topping(models.Model):

name = models.CharField(max\_length=30)

class Pizza(models.Model):

toppings = models.ManyToManyField(Topping)

name = models.CharField(max\_length=30)

In SQLAlchemy we have to construct a helper table to join over:

from sqlalchemy import Column, Integer, String, ForeignKey, Table

from sqlalchemy.orm import relationship, backref

pizza\_toppings = Table('pizza\_toppings', Base.metadata,

Column('topping\_id', Integer, ForeignKey('toppings.id')),

Column('pizza\_id', Integer, ForeignKey('pizzas.id'))

)

class Topping(Base):

\_\_tablename\_\_ = 'toppings'

id = Column(Integer, primary\_key=True)

name = Column(String(30))

class Pizza(models.Model):

\_\_tablename\_\_ = 'pizzas'

id = Column(Integer, primary\_key=True)

name = Column(String(30))

toppings = relationship('Topping', secondary=pizza\_toppings, backref=backref('pizzas', lazy='dynamic'))

## Translating Queries From Django To SQLAlchemy

So this here assumes that you are using scoped sessions like Flask-SQLAlchemy does and unmodified Django. The first example is always how the equivalent Django code looks like and how you would do that with SQLAlchemy:

### Inserting Entries

Inserting entries in Django can be done with either creating an instance of a model or by using the create() method of the object manager:

foo = MyModel(field1='value', field2='value')

foo.save()

# or alternatively

foo = MyModel.objects.create(field1='value', field2='value')

In SQLAlchemy you need to do this instead:

foo = MyModel(field1='value', field2='value')

session.add(foo)

But with that you have only added the object to the session, at that point it has not yet committed the transaction. This has to be done explicitly by yourself when you are happy with all the changes:

session.commit()

### Deleting Entries

Deleting works very much like saving in Django. You get your object and then call the delete() method on it:

obj = MyModel.objects.filter(pk=the\_id).get()

obj.delete()

In SQLAlchemy that operation is performed via the session:

obj = MyModel.query.get(the\_id)

session.delete(obj)

Again, remember to commit your session.

### Updating Entries

How do you update an entry? Just get the object, modify it and commit the session:

obj = MyModel.query.get(the\_id)

obj.name = 'New Value'

session.commit()

### Primary Key Queries

Queries is where Django and SQLAlchemy are the most different. Django uses keyword arguments to the query functions to filter the query, SQLAlchemy generally uses expressions composed out operator objects.

Query by primary key in Django:

obj = MyModel.objects.get(pk=the\_id)

And in SQLAlchemy:

obj = MyModel.query.get(the\_id)

Note that get() returns *None* if the primary key does not exist in SQLAlchemy and will raise a *DoesNotExist* exception in Django.

Generally the get() method is a shortcut in SQLAlchemy that will also not issue a query for that object if it was already queried for that session before. Also unlike Django your primary key can be of any type or be a compound of more than one column.

### General Query Syntax

If you want to filter a query in Django you generally use keyword arguments in the format column\_\_operation=value. For instance column\_\_contains='e' to check if a string column named *column* contains the letter “e”. In SQLAlchemy instead you are using expressions. These expressions can be printed to see what query they would generate.

Here some examples:

>>> print MyModel.id == 23

model.model\_id = :model\_id\_1

>>> print MyModel.id.in\_([1, 2, 3])

model.model\_id IN (:model\_id\_1, :model\_id\_2, :model\_id\_3)

>>> print MyModel.name.contains('e')

model.name LIKE '%%' || :name\_1 || '%%'

Note that SQLAlchemy shows you the placeholders there because it will let the database insert those values later.

The whole expression language expresses pretty much everything that SQL has to offer:

>>> print MyModel.thread\_count + MyModel.post\_count + 1

(model.thread\_count + model.post\_count) + :param\_1

>>> print MyModel.id.between(1, 10) & MyModel.name.startswith('a')

model.model\_id BETWEEN :model\_id\_1 AND :model\_id\_2 AND

model.name LIKE :name\_1 || '%%'

Now this is a biggie, because this is how you can filter for anything if you pass such an expression to filter():

active\_users\_with\_a\_or\_b = User.query.filter(

(User.name.startswith('a') | User.name.startswith('b')) &

(User.is\_active == True)

).all()

To evaluate a query you have a few choices:

1. first() returns the first result from the query and will also tell the database to perform an implicit LIMIT 1. If more than one result is found you won't know and if none is found you get *None* back.
2. one() is similar to first() but it will not limit the result in any way but perform a sanity check on getting the results. It will raise an *NoResultFound* exception back if it did not found a single row or a *MultipleResultsFound* exception if it got more than one result which indicates a bug on your part.
3. all() just evaluates the whole query and returns each row as a list. Why as a list and not as an iterator? First of all because each object returned is also immediately registered on the session. There are of course ways to bypass that, but unless you have an enormous result count you won't notice, secondly because most Python database adapters don't support streaming results anyways.

Now this is nice and everything, but all that model repetition can be annoying. For as long as you are just comparing a column to a given value you can use the filter\_by() function and pass keyword arguments:

user = User.query.filter\_by(username=username).first()

Multiple arguments are automatically joined with AND.

### Date Based Queries

In Django you can use field\_\_year=2011 to select all entries where the year of a field has a specific value. Underneath what usually happens is that an *EXTRACT* expression is issued. Unfortunately that's hugely database dependent and does not map nicely to a function. Thankfully SQLAlchemy provides a helper for that which automatically does the right thing for each database:

from sqlalchemy.sql import extract

entries\_a\_month = Entry.query.filter(

(extract(Entry.pub\_date, 'year') == 2011) &

(extract(Entry.pub\_date, 'month') == 1)

).all()

Quite a few extractions are possible. The most common ones are month, day, year, hour, minute, second, doy (day of year) and dow (day of week).

### Sorting

In Django if you sort something you do that by calling order\_by() and passing it some strings with the columns to order by:

forwards = MyModel.objects.order\_by('pub\_date')

backwards = MyModel.objects.order\_by('-pub\_date')

While it appears that the same is possible in SQLAlchemy you have to be careful because it only works as SQLAlchemy inserts that text directly into the query. What instead you want to be doing is using the expressions again:

forwards = MyModel.query.order\_by(MyModel.pub\_date)

backwards = MyModel.query.order\_by(MyModel.pub\_date.desc())

And again, any expression works in that situation, so you can just easily order by ridiculous expressions if you want.

### Aggregates

Aggregates in Django are a quite new feature and generally not all that awesome, so we're skipping the Django part here. Thankfully they are much better supported in SQLAlchemy as SQLAlchemy just handles them by querying over arbitrary expressions. Functions on the database can be expressed by sqlalchemy.func.functionname in SQLAlchemy. This in combination with arbitrary expressions makes it quite potent. But first the simple case:

from sqlalchemy.sql import func

q = session.query(func.count(User.id))

Now that query obviously does not resolve to a model but a scalar value. In this case if we would call q.first() we would get a single tuple back with a single item: the count. For this case SQLAlchemy provides a nice shortcut: scalar():

>>> session.query(func.count(User.id)).scalar()

1337

What if we want to group by something? Use group\_by() and just iterate over it:

for age, count in session.query(User.age,

func.count(User.id)).group\_by(User.age).all():

print 'Users aged %d: %d' % (age, count)

Distinct counts are simple as well, just call .distinct() on the query. In fact: if you have a rough idea of what the SQL would look like you can get to the expected result with pure guesswork and SQLAlchemy will most likely “just work” ™.

### Joins

Now this is the part where people get constantly confused with SQLAlchemy but fear not, I have you covered. Django hides the business of joins from you. For instance if you want to get all posts written by a specific author that is known by name you would do something like this:

posts = Post.objects.filter( author\_\_name\_\_exact=the\_author\_name)

So how do you do that in SQLAlchemy? The answer is that this means a join is taking place. There are two ways to model that select. First the simple one:

posts = Post.query.join(Author).filter(Author.name == the\_author\_name)

That wasn't too tricky. How does SQLAlchemy know how to do the right thing? It looks at what joins are possible and if only one is, it selects the right one. Alternatively you can explicitly provide what to join on as an expression as second argument to join(). Again, you can get arbitrarily complex there. Everything after the join automatically operates on the last .join()-ed model. If you want to further filter the former model (here *Post*) you can either move them before the .join() call or use .reset\_joinpoint().

Alternatively you could also express this as a subselect:

author\_query = Author.query.filter(Author.name == the\_author\_name)

posts = Post.query.filter(Post.author\_id.in\_(author\_query))

Why does SQLAlchemy not do what Django does? Well, first of all explicit is better than implicit: you know exactly what happens. A regular join is not always what you want or SQL would not provide an outerjoin which of course you can use with SQLAlchemy as well. Secondly, it's really easy to replicated. If you are curious of how that works you can have a look at this subclass of the builtin query that implements Django's filtering with keyword arguments: [sqlalchemy-django-query](https://github.com/mitsuhiko/sqlalchemy-django-query/).

## Why Consider SQLAlchemy?

This article did not really give you any reasons to use SQLAlchemy, did it? But the simple cases is not where SQLAlchemy shines. It's the more complex situations which you can't do at all in Django that work nicely in SQLAlchemy. Oh, and SQLAlchemy does not override all your columns when you just changed one on update ;-)

# Web Server Gateway Interface

The recent [Python 2.5 release](http://www.python.org/download/releases/2.5/) features the addition of the Web Server Gateway Interface Utilities and Reference Implementation package ([wsgiref](http://docs.python.org/lib/module-wsgiref.html)) to Python's standard library.

In this two-part article, we will look at what the Web Server Gateway Interface is, how to use it to write web applications, and how to use middleware components to quickly add powerful functionality. Before diving into these topics, we will also take a brief look at why the specification was created in the first place.

## The Many Frameworks Problem

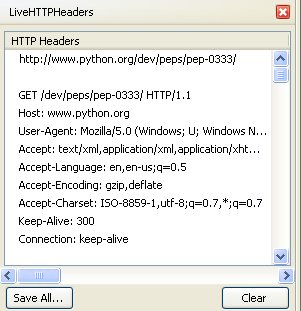
Python is a great language for web development. It is straightforward to learn, has a broad and powerful standard library, and benefits from an active community of developers who maintain a range of XML and database tools, templating languages, servers, and application frameworks. In 2003, when the Web Server Gateway Interface specification was drawn up, the Python community also had one major problem. It was often easier for developers to write their own solutions to web-development problems from scratch rather than reusing and improving existing projects. This resulted in a proliferation of largely incompatible web frameworks. If developers wanted a full and powerful solution, they could use Zope. Developers within the Python community quickly recognized that it would be preferable to have fewer and better-supported frameworks, but since each framework had its strengths and weaknesses at the time, none stood out as a clear candidate for adoption. In the Java world, the servlet architecture meant that applications written with one framework could run on any server supporting the servlet API. The Web Server Gateway Interface (often written WSGI, pronounced "whiskey") was designed to bring the same interoperability that the Java world enjoyed to Python, and to go some way toward unifying the Python web-framework world without stifling the diversity. The full specification is defined in [PEP 333](http://www.python.org/dev/peps/pep-0333/). (PEP is an acronym for Python Enhancement Proposal.) The abstract of the PEP sums up the specification's goals very clearly, stating:

*"This document specifies a proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers."*

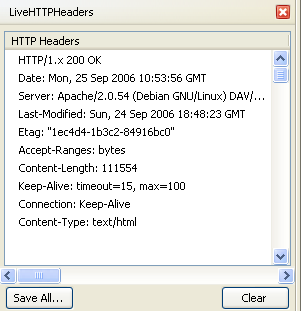
Most Python web frameworks today have a WSGI adapter, and most server technologies (Apache, mod\_python, FastCGI, CGI, etc.) can be used to run WSGI applications, so the vision of web-application portability is fast becoming a reality. The separation of server from application has the clear benefit that application developers supporting the API do not have to worry about how their applications are served, and server developers need only create one interface to support the majority of Python web applications. Simply supporting the WSGI is enough for server and application developers to guarantee a large degree of interoperability. In the next sections, we will look at how to develop and deploy WSGI applications; in Part II of this article, we will look at how to use middleware components to provide facilities such as session handling, interactive debugging, and much more.

## The HTTP Protocol

The Web is largely built on the HTTP protocol, so to understand the WSGI, it is essential to understand how the Web works at the HTTP protocol level. It is very useful to be able to see the HTTP information being sent back and forth between a web browser and a server. One good tool for doing this is the [LiveHTTPHeaders extension](http://livehttpheaders.mozdev.org/) for the [Firefox web browser](http://www.mozilla.com/firefox/) which, when loaded, is visible in the sidebar and displays the HTTP information sent and received on each request. Once it's installed, you can select View->Sidebar->LiveHTTPHeaders from the menu to load the extension in the sidebar. When you request a page, the browser sends an HTTP request to the server. When the server receives that request, it will perform some action (typically running an application or serving a file) and return an HTTP response. Below is the HTTP information sent when visiting the [PEP 333](http://www.python.org/dev/peps/pep-0333/) page:

  
*Figure 1. HTTP Request*

and here is the response returned:

  
*Figure 2. HTTP Response*

As you can see, the browser sends quite a lot of information to the server. The server makes this and other information available to the application. The application then returns a status code (in this case, 200 OK) along with any HTTP headers it wishes to send back. Of particular note is the Content-type header, which tells the browser what sort of content is going to be sent (in this case, text/html). Finally, the application returns the content that will be displayed by the browser (in this case, the HTML that makes up the page). The server may add extra HTTP headers or perform other modifications before the response is returned.

## Hello World!

Here is a simple CGI application that would produce a Hello World! message in the browser:

print "Content-Type: text/html\n\n"

print "<html><body>Hello World!</body></html>"

and here is the same application as a WSGI application:

def application(environ, start\_response):

start\_response('200 OK',[('Content-ype','text/html')])

return ['<html><body>Hello World!</body></html>']

At first glance, this looks a little complicated, so let's think about what information a server needs from an application and vice versa. First of all, an application needs to know about the environment in which it is running. In a CGI application, you can obtain this information with this code:

import os

environ = os.environ

In a WSGI application, this information is supplied directly by the WSGI server as the first parameter to the application callable; in our example above, this parameter is named environ. The server also needs to know the status and headers to set before any content is returned to the browser, so it supplies a second argument to the application -- a callback function taking the status and a list of tuple pairs of headers as arguments. In our example, it is named start\_response. Our application calls start\_response() to set a status of 200 OK, which means everything went fine, and to set the Content-type header to text/html. Finally, the server needs to know the content to return to the browser. Rather than requiring an application to return all the content in one go, the server iterates over the application, returning data to the browser as the application returns it. In our example, this result is achieved by returning a simple list of strings containing the HTML to display the Hello World! message.

In summary then, a WSGI application is any callable (in our case, a simple function) taking an environment dictionary and start\_response callable as positional parameters. The application should call start\_response() with a status code and a list of tuple pairs of headers before it returns an iterable response (in our case, a list with just one string). In normal circumstances, applications can only call start\_response() once; after all, it wouldn't make a lot of sense to start the response twice.

These requirements for applications can also be met by *generators*, classes that override the \_\_call\_\_() method or the \_\_iter\_\_() method. It is possible to use these as WSGI applications instead of following the example above and using a simple function.

The WSGI specification also discusses a third parameter to start\_response() named exc\_info, used in error handling, and a writable object returned by start\_response(), used for backward compatibility -- not for use in new applications or frameworks. (We do not need to worry about these details, but they are mentioned for completeness).

**Testing the Application**

We will start by running our test application as a CGI script. Create a new file named test.py and add the following to it:

def application(environ, start\_response):

start\_response('200 OK',[('Content-type','text/html')])

return ['<html><body>Hello World!</body></html>']

from wsgiref.handlers import CGIHandler

CGIHandler().run(application)

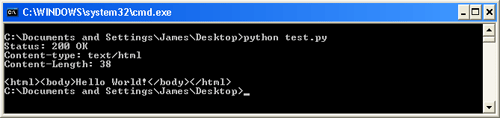
If you are running a version of Python prior to 2.5, you will need to [download](http://python.org/pypi/wsgiref) and install the wsgiref package. You can do this by extracting the wsgiref-0.1.2.zip file (or most recent update) and executing the command:

> python setup.py install

You can then run the test application from the command line to see what information it outputs:

> python test.py

You will see the following:

[](https://www.xml.com/pub/a/2006/09/27/graphics/test_output.png)  
*Figure 3. Test output (Click for full-size image).*

Notice that the CGIHandler has acted the way any other CGI server would, and added the Status and Content-Length to the output. Of course, one of the original motivations for the WSGI was to run the same application on multiple WSGI servers without modification. As an example, this is how you would serve the same application using FastCGI instead of CGI, using the [flup](http://python.org/pypi/flup) package:

from flup.server.fcgi import WSGIServer

WSGIServer(application).run()

You can also run this same, unmodified application on **all** other [WSGI-compliant servers](http://wsgi.org/wsgi/Servers).

## The environ Dictionary

We've seen how to write a simple WSGI application and deploy it as a CGI script or use it with FastCGI, but to write any real application, we'll need access to the environment. This is provided by the first parameter passed to the application by the server, the environ dictionary. The environ dictionary contains all the information about the environment and the request that the application needs. Below is an application named show\_environ that displays a list of all the keys in the environ dictionary:

def show\_environ(environ, start\_response):

start\_response('200 OK',[('Content-type','text/html')])

sorted\_keys = environ.keys()

sorted\_keys.sort()

return ['<html><body><h1>Keys in <tt>environ</tt></h1><p>','<br/>'.join(sorted\_keys),'</p></body></html>',

]

from wsgiref import simple\_server

httpd = simple\_server.WSGIServer(('',8000),

simple\_server.WSGIRequestHandler,

)

httpd.set\_app(show\_environ)

httpd.serve\_forever()

This example serves the application using a standalone server. If you run this example, you will be able to see the output from the application by visiting *http://localhost:8000*. You will notice that environ contains all the usual keys you would expect to see in a CGI environment, such as PATH\_INFO, REMOTE\_ADDR, etc. It also contains the Web Server Gateway Interface keys wsgi.errors, wsgi.file\_wrapper, wsgi.input, wsgi.multiprocess, wsgi.multithread, wsgi.run\_once, wsgi.url\_scheme, and wsgi.version, which provide information about the WSGI environment. The use of these variables is described in the [environ variables](http://www.python.org/dev/peps/pep-0333/#environ-variables) section of the specification. The wsgi.errors key contains an output stream (filelike object). Any information written to the output stream is sent by the server to its error log. This can be useful for debugging purposes, as well as issuing errors or warnings, and it can be used like this:

environ['wsgi.errors'].write('This will be sent to the error log')

Adding this line to the example above, restarting the server, and refreshing the page outputs the message to the console as expected:

[](https://www.xml.com/pub/a/2006/09/27/graphics/console.png)  
*Figure 4. Console (Click for full-size image).*

From the information in environ, you can build any web application, and tools already exist to simplify common tasks, such as extracting the variables submitted in a form from the QUERY\_STRING or rebuilding the original URL. Have a look at [Paste](http://pythonpaste.org) for the lower-level tools or [Pylons](http://pylonshq.com) for a full WSGI framework.

The environ dictionary can actually contain one other type of key: a key defined by the server or a middleware component. Such keys are named using only lowercase letters, numbers, dots, and underscores, and are prefixed with a name that is unique to the defining server or middleware. In Part II of this article, we will look at WSGI middleware and how to add it to existing applications. We will also see an example of a middleware component that adds a key to the environ dictionary to provide session functionality to an application.

## Summary

n Part I we discussed how the Web Server Gateway Interface (WSGI) is helping to unify the Python web-framework world by providing a standard API through which different web applications can operate with different servers. We also looked at the HTTP protocol and how to write and deploy WSGI applications. In the second part of this article, we'll look at how to make use of existing middleware components to add functionality to your WSGI applications.

## What is Middleware?

WSGI middleware is software that behaves like a server to an application, passing an environ dictionary and start\_response callable to the application in the same way a server would. Middleware components also expect the application to return an iterable, such as a list of strings, or to be iterable themselves. Importantly, middleware also behaves like an application to the server, expecting to receive an environ dictionary and start\_response callable itself, and returning an iterable back to the server.

Middleware effectively sits between a server and an application, isolating one from the other, and can therefore do any of the following or a combination thereof:

* Provide more functionality by adding key to environ dictionary
* Change the status
* Intercept an error
* Add, remove, or change headers
* Change a response

Middleware is therefore extremely powerful and can build a broad range of discrete components that can be used with different WSGI servers and applications. For example, a middleware component can:

* Produce error documents when certain status codes are received (typically responding to 404 and 500 codes)
* Email error reports to a developer if a problem occurs
* Provide interactive debugging facilities
* Forward requests to other parts of the application
* Test API compliance of applications and servers to the WSGI
* Authenticate a user
* Cache pages
* Provide a session store
* Gzip the response

Have a look at the [middleware and utilities](http://wsgi.org/wsgi/Middleware_and_Utilities) page on the [wsgi.org](http://wsgi.org) site to get an idea of some of the middleware that already exists. [Paste](http://pythonpaste.org), one of the packages mentioned, contains many middleware components of its own that are not listed separately on the *wsgi.org* page, but are worth taking the time to investigate.

## Getting Started

As an example, we'll create an application that uses session middleware to store the value of a variable between requests. Here is the application that needs session support:

def application(environ, start\_response):

session = environ['beaker.session']

if not session.has\_key('value'):

session['value'] = 0

session['value'] += 1

session.save()

start\_response('200 OK', [('Content-type', 'text/plain')])

return ['The current value is: %d' % session['value']]

The application stores a variable called value in the session store. On each request, the variable is incremented and a message stating its current value is returned. For this application to work, the environ dictionary needs to contain the beaker.session key, which is provided by the session middleware from the [beaker](http://python.org/pypi/Beaker) package.

Here's how you would wrap the application in beaker's session middleware:

from beaker.session import SessionMiddleware

application = SessionMiddleware(

application,

key='mysession',

secret='randomsecret',

)

The new application object behaves just like a normal WSGI application. When the combined application and middleware object is called, the SessionMiddleware adds the beaker.session key to the environ dictionary and calls the original application with the modified environ dictionary and start\_response() callable. The application then calls start\_response() as normal and returns an iterable to the middleware. The middleware returns this information to the server so that, from the server's point of view, the combined middleware and application can be treated in exactly the same way as a normal WSGI application.

You can test the example above by serving the finished application with the following code and visiting *http://localhost:8000* on your local machine once the server is running:

def application(environ, start\_response):

session = environ['beaker.session']

if not session.has\_key('value'):

session['value'] = 0

session['value'] += 1

session.save()

start\_response('200 OK', [('Content-type', 'text/plain')])

return ['The current value is: %d' % session['value']]

from beaker.session import SessionMiddleware

application = SessionMiddleware(

application,

key='mysession',

secret='randomsecret',

)

from wsgiref import simple\_server

httpd = simple\_server.WSGIServer(

('',8000),

simple\_server.WSGIRequestHandler,

)

httpd.set\_app(application)

httpd.serve\_forever()

If you are running a version of Python prior to 2.5, you will need to download and install the [wsgiref](http://python.org/pypi/wsgiref) package as described in the Part I of this article. You will also need to download and install the beaker package, which provides SessionMiddleware.

If you test the example above, you may find the count goes from 1 to 3, missing 2. This is because many web browsers try to retrieve a /favicon.ico file the first time a site is visited, and this request also results in value being incremented. If you have the [LiveHTTPHeaders extension](http://livehttpheaders.mozdev.org/) for the [Firefox web browser](http://www.mozilla.com/firefox/) installed, you'll be able to see the request being made when you visit a site not already in the browser's cache.

## Middleware Chains

In the previous example, we saw how adding a single middleware component to an application gave it powerful new functionality. In fact, you don't have to stop at one middleware component. Since a combined middleware and application object is also a valid WSGI application, you can also wrap the combined application and middleware object in another middleware component. This leads to the idea of *middleware chains*, where you have a number of pieces of middleware between the server and the application. Below is an example using some fictional middleware components:

MyEnvironMiddleware(

MyStatusMiddleware(

SessionMiddleware(

application,

key='mysession',

secret='randomsecret',

)

),

'Some Configuration Option',

)

In situations such as the one above where you are using a number of middleware components, it is often more convenient to structure your code like this:

application = SessionMiddleware(application, key='mysession', secret='randomsecret',)

application = MyStatusMiddleware(application)

application = MyEnvironMiddleware(application, 'Some Configuration Option')

In a similar way, it's possible to create an entire web-framework stack just out of individual WSGI middleware components; indeed, the popular [Pylons](http://pylonshq.com) web framework, used to build production sites worldwide, already takes this approach. Having a stack made entirely from WSGI middleware has a huge advantage: developers are free to pick and choose the components they need, or even to replace the parts of the stack they don't like by simply changing which middleware they use. If you've ever tried changing parts of the application stack in other framework architectures, you understand how hard it can sometimes be.

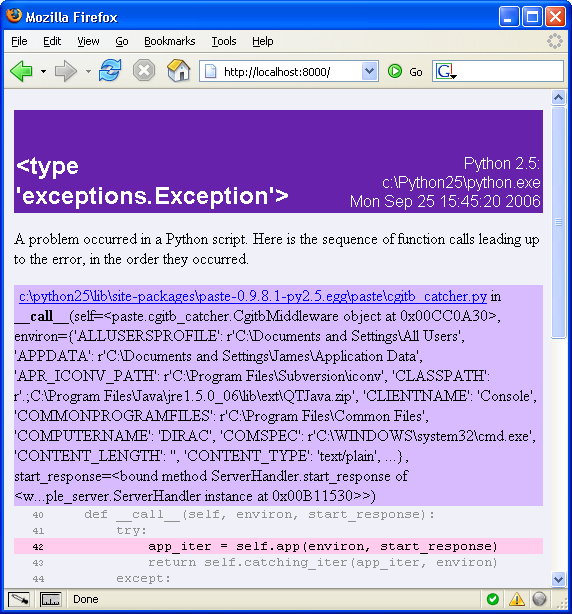
## Error Handling

During development of an application, it's useful to be able to debug errors. The first step is to display an error report. To do this, you can use the CgitbMiddleware middleware from the Paste project:

from paste.cgitb\_catcher import CgitbMiddleware

application = CgitbMiddleware(application, {'debug':True})

Now if an exception is raised in your application code, a full error report--similar to one shown below--will be displayed.

  
*Figure 1. CgitbMiddleware middleware in action*

For production deployment, you'd want to disable this facility to prevent a visitor from accidentally being shown the values of important variables, such as passwords that might otherwise be displayed if an error occurred.

While it is undoubtedly useful to have a traceback report, it would be even more useful to be able to interactively debug each part of the call stack up to the point at which the error occurred by using a web-based command prompt. Such a solution already exists and can be added in exactly the same way:

from paste.evalexception import EvalException

application = EvalException(application)

The EvalException middleware will not work in a multiprocess environment, such as a WSGI application deployed as a CGI script, because the middleware must be able to store information about the error in memory. This is not possible if the whole application is restarted on each request. If you want to test the EvalException middleware, you could use this code:

def application(environ, start\_response):

start\_response('200 OK',[('Content-type','text/plain')])

response = []

variable1 = 'All local variables will be displayed'

response.append('Everything is going fine...\n')

raise Exception('Something went wrong!')

response.append("We won't get to here!")

return response

from paste.evalexception import EvalException

application = EvalException(application)

from wsgiref import simple\_server

httpd = simple\_server.WSGIServer(

('',8000),

simple\_server.WSGIRequestHandler,

)

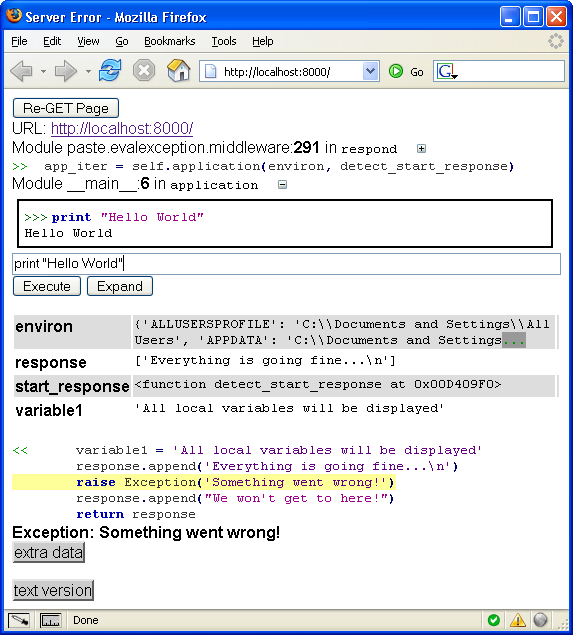
httpd.set\_app(application)

httpd.serve\_forever()

If you run the example above and visit *http://localhost:8000*, you will see the error report. Clicking on the plusicon will give you the interactive debug prompt, and clicking on >>will give you a representation of the code at the point that the error occurred. Try entering this at the prompt and press enter:

print "Hello World!"

You will see the Hello World! printed exactly as if it were entered at a normal Python prompt because the middleware acts like a full Python interpreter. You can even use the up and down arrows on your keyboard to scroll through the command history, just as you can at a real Python prompt. The EvalException middleware also displays the values of local variables and has an extra data button that displays extra information about the environment. The screenshot below shows the middleware in action with the interactive prompt and local variables, including variable1, displayed:

  
*Figure 2. EvalException middleware in action*

Having this much power makes it much easier to debug applications, but makes it even more important that you disable debugging for production use so that malicious visitors can't execute destructive commands through your debug screen if an error occurred. Once again, this example shows just how much useful functionality can be added to an application using a single middleware component.

## Configuration

When the Web Server Gateway Interface was being drawn up, there were a number of discussions about how best to deploy a finished application with middleware. Clearly developers couldn't expect non-technical users to directly modify the middleware chains themselves. The most widely adopted solution is to use [PasteDeploy](http://pythonpaste.org/deploy/). Users configure a *config* file in a familiar INI-style format, and the desired application--with any necessary middleware--is created by PasteDeploy from that file. PasteDeploy is used like this:

from paste.deploy import loadapp

application = loadapp('config:/path/to/config.ini')

The configuration file can be used to specify server settings, including middleware, and even combine multiple WSGI applications together into a composite application. All the options are well documented on the PasteDeploy site.

## The Many Frameworks Problem Revisited

At the start of Part I of this article, we looked at how the WSGI was created to help solve the fragmentation of the Python web community. The WSGI specification also had wider ambitions. [PEP 333](http://python.org/dev/peps/pep-0333/) states:

*"If middleware can be both simple and robust, and WSGI is widely available in servers and frameworks, it allows for the possibility of an entirely new kind of Python web application framework: one consisting of loosely coupled WSGI middleware components. Indeed, existing framework authors may even choose to refactor their frameworks' existing services to be provided in this way, becoming more like libraries used with WSGI, and less like monolithic frameworks. This would then allow application developers to choose "best-of-breed" components for specific functionality, rather than having to commit to all the pros and cons of a single framework."*

With today's powerful middleware, this vision is fast becoming a reality. Emerging projects such as [Clever Harold](http://www.cleverharold.com/) provide just such a framework of loosely-coupled middleware components. Projects such as Pylons go further still, providing a ready-made configuration of WSGI middleware. We have also seen existing projects like Myghty refactored to work better with WSGI configurations.

The WSGI has shifted the point of reuse from the framework itself to individual middleware components. While developers can still create their own solutions to web development problems, as long as they're creating, using, and improving middleware components, the whole Python community now benefits.

I hope this article has demonstrated some of the power of WSGI middleware and will encourage you to make use of the specification and the various projects that already implement it. Here are some useful places to start if you wish to learn more about WSGI programming.

* [PEP 333](http://python.org/dev/peps/pep-0333/)
* [Web Server Gateway Interface website](http://wsgi.org)
* [Web-SIG mailing list](http://python.org/community/sigs/current/web-sig/)
* [Paste website](http://pythonpaste.org/)
* [Pylons website](http://pylonshq.com)
* [Example code](https://www.xml.com/pub/a/2006/10/04/examples/code.zip)

1. The **Web Server Gateway Interface (WSGI)** is a simple calling convention for web servers to forward requests to web applications or frameworks written in the Python programming language. The current version of WSGI, version 1.0.1, is specified in Python Enhancement Proposal (PEP) 3333. [↑](#footnote-ref-1)