Creating RESTful Apps with Django

John Strickler

Version 1.0, August 2023

Table of Contents

About Creating RESTful Apps with Django

© 2023 CJ Associates *(Licensed to Protech)*

Course Outline

Day 1

Day 2

Day 3

Day 4

NOTE The actual schedule varies with circumstances. The last day may include *ad hoc* topics requested by students

© 2023 CJ Associates *(Licensed to Protech)*

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **pydjrest**. See the setup guides for details.

What's in the files?

pydjrest contains all files necessary for the class
pydjrest/EXAMPLES/ contains the examples from the course manuals.
pydjrest/ANSWERS/ contains sample answers to the labs.
pydjrest/DATA/ contains data used in examples and answers
pydjrest/SETUP/ contains any needed setup scripts (may be empty)
pydjrest/TEMP/ initially empty; used by some examples for output files

The following folders *may* be present:

pydjrest/BIG_DATA/ contains large data files used in examples and answers
pydjrest/NOTEBOOKS/ Jupyter notebooks for use in class
pydjrest/LOGS/ initially empty; used by some examples to write log files

NOTE	The student files do not contain Python itself. It will need to be installed separately. This may already have been done.

Examples

Most of the examples from the course manual are provided in EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

```
import sys   # Import the sys module

print(sys.argv) # Print all parameters, including script itself

name = sys.argv[1]  # Get the first actual parameter
print("name is", name)
```

cmd_line_args.py apple mango 123

```
['/Users/jstrick/curr/courses/python/common/examples/cmd_line_args.py', 'apple', 'mango',
'123']
name is apple
```


© 2023 CJ Associates *(Licensed to Protech)*

Appendices

Classroom etiquette

Remote learning

- Mic off when you're not speaking. If multiple mics are on, it makes it difficult to hear
- The instructor doesn't know you need help unless you let them know via voice or chat.
- It's ok to ask for help a lot.
  - Ask questions. Ask questions. Ask questions.
  - ***INTERACT*** with the instructor and other students.
- Log off the remote S/W at the end of the day

In-person learning

- Noisemakers off
- No phone conversations
- Come and go quietly during class.

Please turn off cell phone ringers and other noisemakers.

If you need to have a phone conversation, please leave the classroom.

We're all adults here; feel free to leave the clasroom if you need to use the restroom, make a phone call, etc. You don't have to wait for a lab or break, but please try not to disturb others.

IMPORTANT	Please do not bring any exploding penguins to class. They might maim, dismember, or otherwise disturb your fellow students.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 1: Using Visual Studio Code with Python

Objectives

- Use VS Code to develop Python applications

About Visual Studio Code

Visual Studio Code (AKA "Code" or "VS Code") is a full-featured IDE for programming. It is free, lightweight, and works across common platforms and with many languages.

Visual Studio Code Features

- Autocomplete (AKA IntelliSense[tm])
- Autoindent
- Syntax checking/highlighting
- Debugging
- git integration
- Code navigation
- Command palette (easily find any command)
- Smart search-and-replace
- Project management
- Split views
- Zen mode (hide UI details)
- Code snippets (macros)
- Variable explorer
- Integrated Python console
- Interpreter configuration
- Unit testing tools
- Keyboard shortcuts
- Many powerful extensions
- Works with many languages
- Free

© 2023 CJ Associates *(Licensed to Protech)*

Getting started

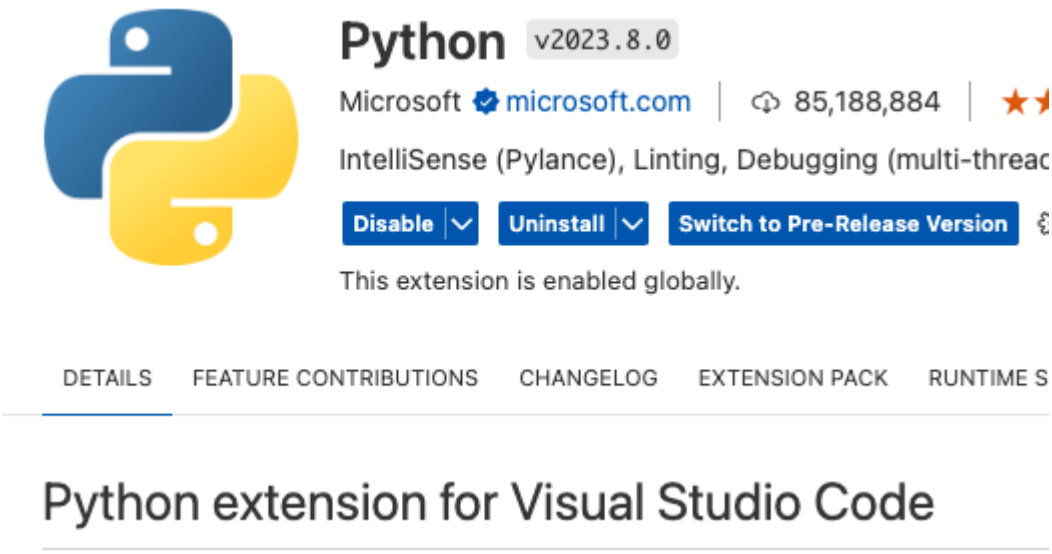
Installing

If VS Code is not already installed, download and install from https:/code.visualstudio.com .

Adding the Python extension

This class requires the Microsoft extension for working with Python. If it is not installed, please follow these steps:

1. Start VS Code
2. Go to **Extensions** in sidebar on left
3. Search for "python"
4. Select and install the Python extension from Microsoft



Configuration

Many aspects of VS Code can be configured. A little configuration makes VS Code more convenient.

To configure settings, go to **File › Preferences › Settings**.

NOTE	On Macs, start with the **Code** menu rather than the **File** menu.

Auto save

To turn on Auto Save, which saves your file as you type:

1. Search for "auto save"
2. change **Files: Auto Save** to **afterDelay**.

Launch folder

For convenience, set VS Code to run a script from the folder that contains it.

1. Search for "execute in"
2. Check the box for **Python › Terminal: Execute in File Dir**

Minimap

If you do not want to use the minimap (the narrow guide strip along the right margin), you can turn it off.

1. Search for "minimap enabled"
2. Uncheck **Editor › Minimap: Enabled**

Editor font size

To change the font size of the code editor window:

1. Search for "editor font size"
2. Set **Editor: Font Size** to desired value

Terminal font size

To change the font size of the terminal window (bottom pane where scripts are executed):

1. Search for "terminal font size"
2. Set **Terminal › Integrated: Font Size** to desired value

© 2023 CJ Associates *(Licensed to Protech)*

Themes

To change the overall theme (colors, font, etc):

1. Go to **File › Preferences › Theme › Color Theme**
2. Choose a new theme

> **NOTE** **Visual Studio Code** may be already setup and configured.

Opening a project

To open a folder, choose **File › Open Folder...**. When you open a folder, it automatically becomes a VS Code *workspace*. This means that you can configure settings specific to this folder and any folders under it.

Generally speaking, this is great for managing individual projects.

Creating a Python script

Use the `File` menu

Go to **File › New File...**.

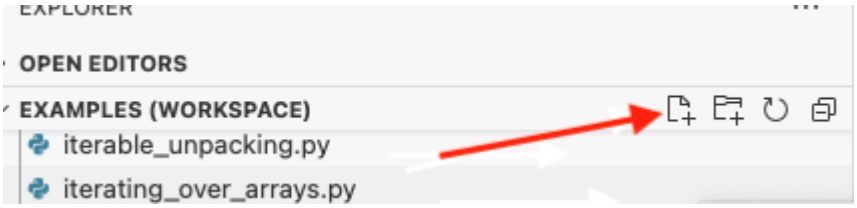
Type a name in the blank, including the `.py` extension. You'll get a file dialog for where to save it. The default location will be the same folder as the currently open file, so be sure to make sure you save it in the desired folder.

TIP

You can also just select "Python File" and Code will create a new editor window named `untitled-n`, where *n* is a unique number. You can then use **File › Save** to save the file with a permanent name.

Use the "new file" icon

Click on the "new file" icon next to the folder/workplace name in the Explorer panel.



Click on "select a language" and choose Python. As above, it will create a new file named `untitled-n`. Use **File › Save** to save the file.

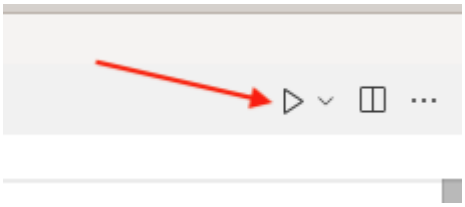
This will create a file in the same folder as the currently open file.

© 2023 CJ Associates *(Licensed to Protech)*

Running the script

Click the "run" icon

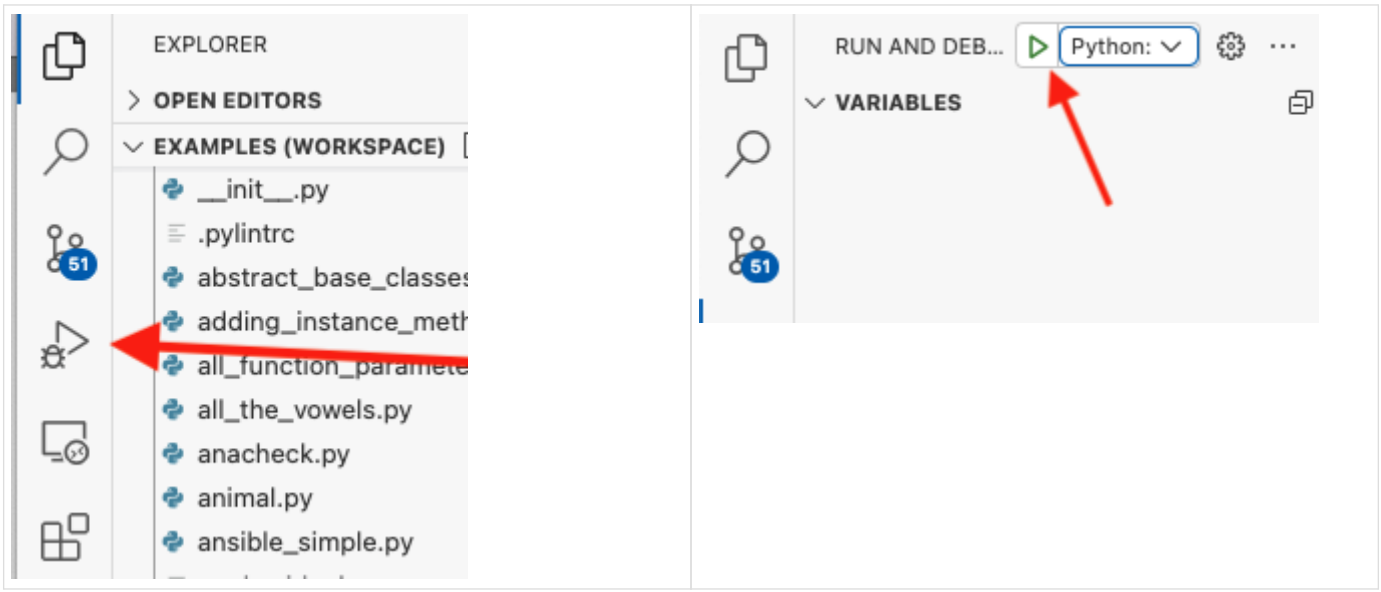
The easiest way to run a script is to click the "Run Python File" icon next to the editor tabs



Once you have run the file, you can re-run it by pressing up-arrow in the terminal window. This makes it easy to add arguments.

Use Run/Debug panel

You can open the Run/Debug panel, and then click on the green arrow to run the script. This uses the default launch configuration, which can be customized.



TIP You can create custom launch configurations to fine-tune how you want to launch each script.

Use Run menu

The Run menu is essentially a shortcut for the tools on the Run/Debug panel. You can choose to run with or without debugging. If you have no breakpoints set, there is not much difference.

Chapter 1 Exercises

Exercise 1-1 (hellocode.py)

In VS Code, create a new script named `hellocode.py`. Put one or more `print()` calls in it and run it with VS Code

Now run it from the command line as well (not in the VS Code integrated terminal).

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 2: Django Overview

Objectives

- Learn what Django is, and what it can do
- See what files are generated by Django
- Understand the difference between projects and apps

What is Django?

- Full-featured web framework
- Many extensions
- Supports web apps and web services
- Builds basic app, you fill in details

Django is a complete framework for implementing all kinds of interactive web applications, including web services. The default install includes many subpackages to handle all aspects of web development, and there are extensions for less-common needs.

Django provides the tools and components to rapidly create a web app, storing data in any popular database, and using templates to generate HTML. An admin interface to your database is included in the basic framework.

Being a framework, Django creates the fundamental project structure, with Python scripts ready to fill in with your details. Configuration is handled by several predefined scripts.

The rationale for Django is to handle the tedious parts of Web development, such as DB admin, state, sessions, security, etc., and let the developers focus on the domain-specific part of their apps.

Developers should not have to reinvent any wheels when using Django. Django scales easily from a tiny app to global commercial websites.

© 2023 CJ Associates *(Licensed to Protech)*

Django features

- Models (Object Relational Mapper)
- Views (functions that render templates or data)
- Controllers (implicit in Django architecture)
- Web server for testing
- Form handling
- Unit testing
- Caching framework
- I18N
- Serialization (very handy for web services!)
- Admin interface
- Extensible authentication system
- Support for individual sites sharing apps
- Protection from XSS, SQL injection, password cracking, etc.

> The motto from Django's home page is "The Web framework for perfectionists with deadlines"

Who created Django?

- Created at Lawrence World-Journal
- Named for Django Reinhardt
- First developed 2003
- Released publicly 2005
- Released to Django Software Foundation 2008

Django was created in 2003 by programmers at the Lawrence World-Journal newspaper. They had become frustrated with PHP, and had their own ideas about how a web framework should be implemented.

Django was released publicly under the BSD software license in 2005. It has quickly grown in popularity.

Django is now maintained by the non-profit Django Software Foundation.

Django Reinhardt was a well-known guitarist who was active in the Paris jazz scene in the 1930s and 1940s.

© 2023 CJ Associates *(Licensed to Protech)*

Django in a nutshell

- Create project and app
- Build model(s)
- Define views
- Design templates
- Map URLs

Generate the project

It's easy to get started with Django. The first step is to generate a project and an app. You can use the builtin tool `django-admin` or the `cookiecutter` package.

Define models

Assuming you will keep permanent data as part of the application, the next step is to define *models*. These map to tables in the database. The database can be new or existing. If it's existing, then Django can generate models from the database; if it's new, then you can define the models in Python and Django will generate the needed SQL to create the DB.

Design templates

Templates are files that contain the source to a web page – HTML, CSS, and JavaScript – as well as placeholders for app data.

Create views

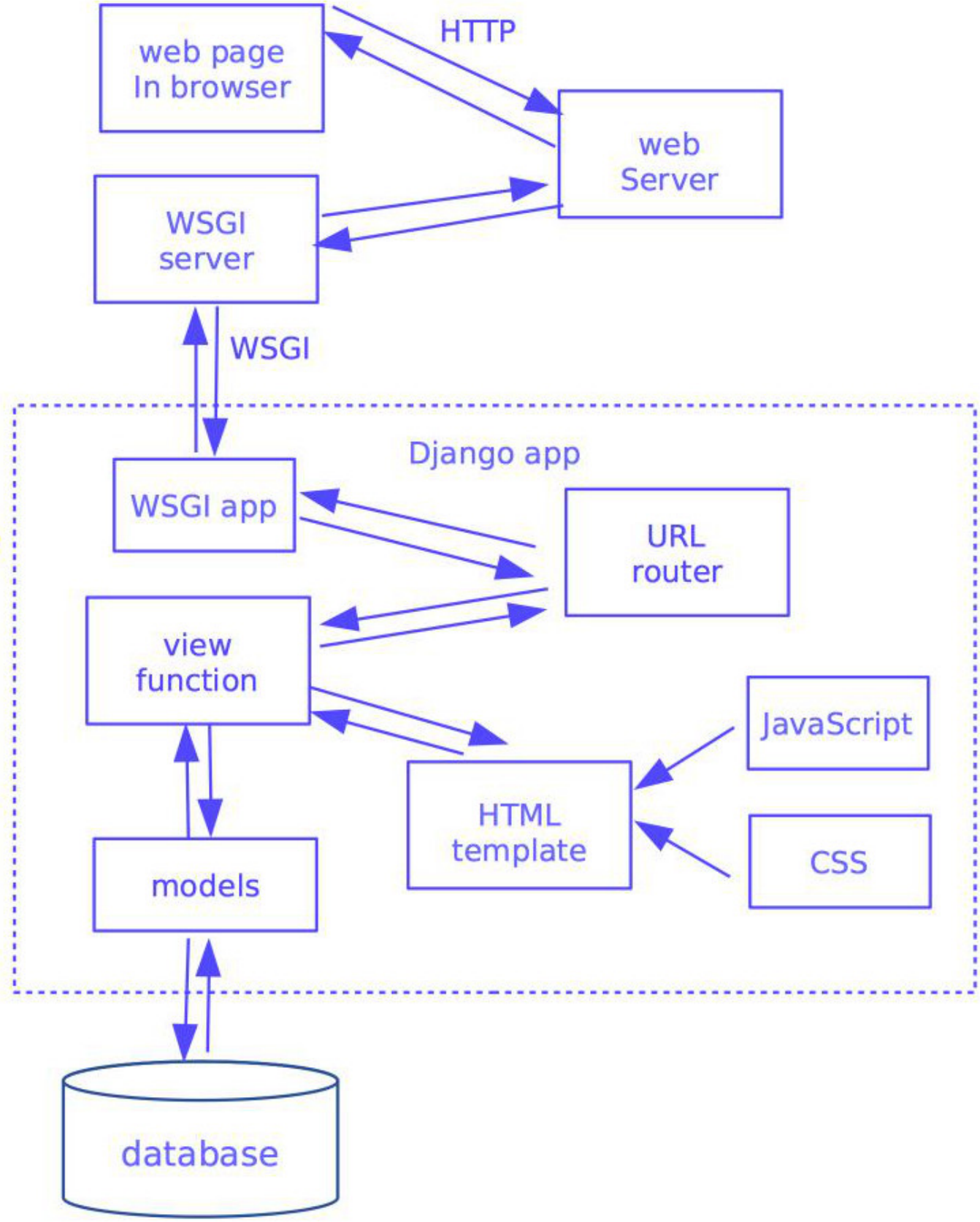
The next step is to create some *view functions*. A view function retrieves data via your models and passes the data and a *template* to a template renderer, which fills in the placeholders with the data and returns the final version of the web page.

Map URLs to views

The last piece of the puzzle is to map URLs in your app to specific view functions. This is done at both the project and app level.

NOTE The four steps above (after creating the project) do not have to be done in any specific order.

Django Architecture



© 2023 CJ Associates *(Licensed to Protech)*

Projects and apps

- Projects are a collection of apps
- Apps are a collection of pages

In Django, the first step is to create a project, and then create one or more apps within the project. While you can set up your project differently, it makes sense to work with Django's defaults.

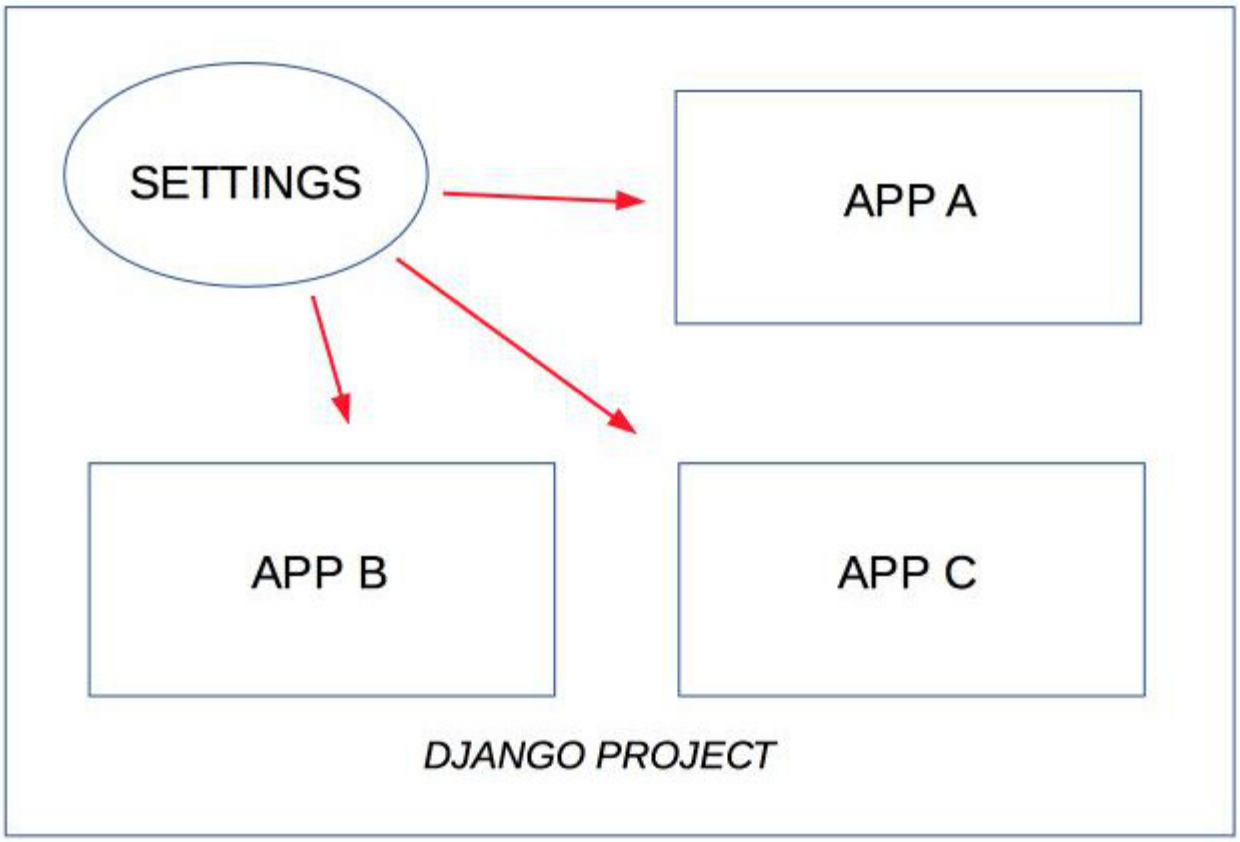
An app is a Python package containing related modules, including models, views, url config, and templates.

A project is a collection of apps that shares the same configuration. This usually means the same database, among other things. While one project may be associated with one web site, Django setup is really very flexible.

All Django requires is that there be a project, and at least one app. Once you learn the ropes, you can even break that rule. The same app can be used in multiple projects.

When you create a project with `django-admin`, it creates some default folders and modules. Then, you can create one or more apps within the project.

NOTE `django-admin` does not create all the modules needed for your project. In particular, it does not create `urls.py` in the app module.



© 2023 CJ Associates *(Licensed to Protech)*

Chapter 3: Getting Started

Objectives

- Learn what a Django site contains
- See example Django configuration
- Build a minimal Django application

© 2023 CJ Associates *(Licensed to Protech)*

What is a site?

- One instance of Django
- Collection of apps
- Shared configuration

A **site** is one instance of Django, running on a specific host (technically, a specific IP address) and on a specific port. One site can contain any number of web apps (apps). A project contains the site, which has shared configuration that all the apps will use. We can refer to a Django **project** as one *programming project* — a folder containing a site, some apps, and other components such as documentation. The project folder contains **manage.py**, a script for managing the site.

This shared configuration includes URL routing, database configuration, middleware, and other settings.

NOTE Some developers use **site** and **project** interchangeably.

© 2023 CJ Associates *(Licensed to Protech)*

Starting a project

Django has a minimum set of files needed for a site. While you *can* write a complete Django app in one file (see **minimalapp.py** in EXAMPLES), this is not recommended. It's easier to maintain a project when code is separated into functional areas. Thus, URL configuration goes in one module, views go in another module, and so on.

While you can create a project manually, most developers use a startup tool. This will create a set of files to get the site started.

The builtin script `django-admin` has a command `startproject`, which will create a simple site layout:

```
django-admin startproject projectname
```

This creates the needed files, including **settings.py** with default configuration values.

If you created a site named **helloworld**, the project files would look similar to this:

```
helloworld              # project folder
├── helloworld          # site package
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py           # site manager
```

Note that there is a package with the same name as the site under the top-level folder. This package contains settings and configuration for the project.

NOTE If `django-admin` is not in your path, try `python -m django startproject`.

We'll look at `cookiecutter`, a more powerful startup tool, in a later chapter.

manage.py

- Application manager script
- Manages database, server, etc.

manage.py is an admin script created for each app. It is similar to django-admin, but is customized for your site.

It is used to create apps, to manage databases, and to run the development server, among other things.

There are many subcommands. We will not necessarily use all possible commands in this course, but we will use many of them.

The general syntax is

```
python manage.py subcommand [param] ...
```

To get a list of subcommands, use

```
python manage.py help
```

Once you have created a site, you can run the development server immediately. Go to the folder with manage.py, and type

```
python manage.py runserver
```

Go to localhost:8000 in a browser and you should see a default page.

© 2023 CJ Associates *(Licensed to Protech)*

manage.py subcommands

auth
changepassword
createsuperuser

contenttypes
remove_stale_contenttypes

django
check
compilemessages
createcachetable
dbshell
diffsettings
dumpdata
flush
inspectdb
loaddata
makemessages
makemigrations
migrate
sendtestemail
shell
showmigrations
sqlflush
sqlmigrate
sqlsequencereset
squashmigrations
startapp
startproject
test
testserver

sessions
clearsessions

staticfiles
collectstatic
findstatic
runserver

© 2023 CJ Associates *(Licensed to Protech)*

Shared configuration

- Stored in site/site/settings.py
- Installed apps (yours + plugins)
- URL mappings
- Database info
- Password validators
- Time zone info
- Location of static files

The settings.py script in the site module contains overall site configuration.

Among its contents are the installed apps for the site, which includes your apps, as well as any plugin apps provided by Django. There are several such apps provided in the default installation.

There is also top-level configuration for the URLs on your site (from site/site/urls.py). However, URLs are normally configured within each app.

This is where (by default) you specify what database (or databases) the apps in the site will use. In addition, you can specify password validators, time zone info, the location of static files, and many other details.

The settings module does not have to be called settings.py, but it is a convention that many developers follow.

NOTE See all possible settings here: https://docs.djangoproject.com/en/4.0/ref/settings/

© 2023 CJ Associates *(Licensed to Protech)*

Example

```
django-admin startproject zoo
cd zoo
python manage.py startapp wombat
python manage.py startapp koala
```

produces

```
zoo                             # project
├── koala                       # app
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py                   # site manager
├── wombat                      # app
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
└── zoo                         # site
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Steps to create a Django App

- Start app with manage.py (or other tool)
- Create database models
- Design templates
- Implement views
- Add app to settings
- Map urls
- Deploy app

To create a Django app, first create the app using **manage.py startapp**. This will create a minimal app package. You will add modules to it as needed.

Once the app is created, add the app name to **INSTALLED_APPS** in the settings module (default **settings.py**).

The next step is to create one or more *models* in **models**. This is optional if you aren't going to be using Django to access a database. In this chapter we'll skip models for now.

After the models are created, you can create a template. A template is an HTML file with placeholders for your app's data. There is no particular rule for what the templates contain, but typically they display either data for the user to see, or else forms for the user to fill out. We will skip templates for the very simple projects in this chapter

Now it's time to create one or more views in the **views** module, to provide data for the templates. A view is called when a particular URL is requested by the client (browser). The view connects the data to the template, and returns an HTTPResponse with the filled-out template (or other content).

To make views accessible, they must be matched to a particular URL within your application. This is done by adding entries to **urlpatterns** in the **urls** module.

Once the app has been created with startapp, you can do the above steps in any order.

The final step is to deploy the app on a web server. For development, we will use Django's builtin server.

NOTE You can add as many apps to a site as you like.

© 2023 CJ Associates *(Licensed to Protech)*

Creating the app

- Command:

```
manage.py startapp appname
```

- Creates folder (package) for app
- Provides empty modules

To create the app, go to the top level of the site. Issue the command

```
python manage.py startapp appname
```

This will create a folder named *appname*. In it, you will find some default app modules. These are not necessarily all the modules needed, and you don't have to keep those names; however, Django has a lot of "configuration by convention", so it's a good idea not to change names unless you have a good reason.

The app and the site cannot have the same name, because Django automatically creates a module in the site with the same name as the site. This is where the site-wide configuration will be stored. Keep the names simple, as they will also be Python module names that you will use in your code. Names must follow the normal rules of Python identifiers – letters, digits, and underscores. They cannot start with a digit, and most developers avoid underscores in site and app names.

Table 1. Default App Modules and Packages

Module	Description
`admin.py`	general admin settings
`apps.py`	application definition (defaults are OK)
`migrations`	folder for DB migration info
`models.py`	DB definitions
`tests.py`	unit tests
`views.py`	view functions

Register the app

- app object automatically created
- Register in site's settings.py

A minor, but crucial, step is register the app in the site's `settings.py` module.

In the app folder (generated by `manage.py startapp` or **cookiecutter**) will be a module named `apps.py`. In the module will be a config class named with the title-cased version of the app name plus "Config". Underscores are removed from the name.

For example, an app named `car_search` would have a config class named `CarSearchConfig`.

Add the app config class (as a string) to the **INSTALLED_APPS** list in the site-level settings.py that was auto-generated.

NOTE When using the "official" cookiecutter template **cookiecutter-django**, add the app config class to **LOCAL_APPS** in **config/base.py**.

Example

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

TIP Everyone forgets to do this, so do it as soon as you create the app.

© 2023 CJ Associates *(Licensed to Protech)*

Create views

- Return content to browser
- Just normal functions
- Expect request parameter
- Return HTML

The next step is to create one or more view functions. These functions return some content to the browser, typically HTML.

The view functions are just normal Python functions. They are passed the HTTP request object, from which the app can get URL parameters, and other information.

In later chapters, we will render HTML templates, but for simplicity right now we will use the **HttpResponse** class provided by Django.

An HttpResponse takes a string to return, and automatically adds a default HTTP status code and default HTTP headers.

NOTE Other HTTP status codes and headers can be specified by adding parameters to HttpResponse.

Example

djangoexamples/djdemo/greet/views.py

```
from django.http import HttpResponse
from datetime import datetime

# home view  ("index.html" equivalent)
def home(request):
    timestamp = datetime.now().strftime("%I:%M %p on %B %d, %Y")
    return HttpResponse("Hello, Django World at {}".format(timestamp))
```

The request object

- Passed into all views
- Contains information from the HTTP request

All views are passed an **HttpRequest** object as a parameter. This represents in the incoming HTTP request. From this object you can get any of the details about the request.

Some of the information available:

- Original full path of the request
- HTTP headers
- Browser ID string
- Cookies
- User name
- Browser IP

© 2023 CJ Associates *(Licensed to Protech)*

Configure the URLs

- Associate URLs with view functions
- Create urls.py in app
- Configure in site's urls.py

To associate views with particular paths (URLs), you create a url mapper, typically in a file called **urls.py**.

You can map URLs at both the site level (***site/site/*urls.py**) or the app level(***site/app/*urls.py**).

In either case, urls.py contains a list named **urlpatterns** that contains one or more URL mappings.

In Django 1.x, URL mappings will be **url** objects. These objects are initialized with a regular expression, the view function itself, and a label (the "view name") that refers to that particular path. This view name can be used to automatically construct a URL that points to the view from other locations.

In Django 2.x, URL mappings will be **path** objects. They work the same way, but the first argument is just a string, not a regular expression.

The view functions must be imported into urls.py.

© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/djdemo/djdemo/urls.py

```
"""demo URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include
    2. Add a URL to urlpatterns:  path('my_app', include('my_app.urls'),
namespace='my_app')
"""
from django.conf import settings
from django.urls import path, include
from django.contrib import admin

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('greet.urls', namespace='greet')),
]

# include Django Debug toolbar if DEBUG is set
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
    ] + urlpatterns
```


© 2023 CJ Associates *(Licensed to Protech)*

The development server

- Minimal web server
- Do not use for production
- Single-threaded
- Extra debugging support

For convenience while developing apps, Django provides a minimal web server that is integrated into your project. This server does not support multiple clients, and while it is reasonably fast, it is not fast enough for real-world services.

To start the server, you should be in the top-level folder of your project (the one that contains **manage.py**. Type

```
python manage.py runserver
```

It does contain extra debugging support. The manage.py tool described next is used to start the development server.

You should definitely not use the builtin server for production, due to the above and other factors. It is not secure enough for the real world. For production, use NGINX, Apache, IIS, or other proven servers.

© 2023 CJ Associates *(Licensed to Protech)*

Serve app with builtin server

- Use **manage.py runserver**
- http://localhost:8000 (default)

All that's left is to deploy the app. We'll discuss later how to deploy in production, but for development you can use Django's builtin server.

To launch the app, just say

```
python manage.py runserver
```

That's all there is to it.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 3 Exercises

Exercise 3-1 projectone

Using Django's builtin project starter (django-admin), create a new Django project named **djangohello**. Add an app named **hello**. Follow the steps outlined in this chapter. Create a view named **home** which returns the phrase "I am now a Django developer" as an **HTTPResponse.**

Configure the project's urls.py to map the empty URL to the **home** view.

Use **manage.py** to start up the site. Go to a browser, type in **http://localhost:8000**, and view your handiwork.

Chapter 4: Using Cookiecutter

Objectives

- Discover cookiecutter
- Use cookiecutter to start a project

© 2023 CJ Associates *(Licensed to Protech)*

About cookiecutter

- Alternate startup tool
- Creates "real-life" Django setup
- Different from **startproject**
- Authors
  - Audrey Roy Greenfeld
  - Daniel Roy Greenfeld

cookiecutter is a utility written by Audrey and Roy Greenfeld to make it easy to replicate a standard setup for Django. The cookiecutter command prompts you for information, then creates the project folder.

It uses a cookiecutter *template*, which is a folder, to create the new project. There are several templates on **github** to choose from. The standard template, cookiecutter-django, is a bit advanced ("bleeding edge", in the authors' words) for class, so two basic templates (one for projects and one for apps) are provided with the class files.

The utility copies the template layout (all folders and files) to a new folder which is the "slug" (short name) of your project. It inserts your project name in the appropriate places.

Not all templates are templates! When you see 'template' above, it refers to a cookiecutter source folder and its files, *not* a Django or Jinja2 HTML template.

cookiecutter home page: https://github.com/audreyr/cookiecutter
cookiecutter docs: https://cookiecutter.readthedocs.io

Using cookiecutter

- cookiecutter *cookiecutter_name*
- creates folder under current

To use **cookiecutter**, execute the cookiecutter command with one argument, the name of the cookiecutter template folder. This can be a folder on the local hard drive, or it can be online. cookiecutter supports Github, Mercurial, and GitLab repositories, but you can also just give the URL to any online file.

cookiecutter will ask a few configuration questions. (The standard template is much more elaborate). One of the questions is the name of the project "slug". This is the short name that will be used to name files and folders, so it should be short and only contain letters, digits, and underscores. The default slug is created from the project name, but can be anything you like. The project slug will be used as the name of the main project folder.

The new folder, which is your Django project, is created in the current folder.

Example local usage

```
cookiecutter ../SETUP/cookiecutter-{django-version}
project_name [Project Name]: My Wonderful Project
project_slug [my_wonderful_project]: wonderful
project_description [Short Description of the project]: A Wonderful Django Project for
Class
time_zone [America/New_York]:

Congratulations! you have created project wonderful

Now, cd into the wonderful folder.

Execute this command to set up Django's admin and user databases:

    python manage.py migrate

At this point, you can start creating apps.
```


© 2023 CJ Associates *(Licensed to Protech)*

tree wonderful

```
wonderful
├── manage.py
└── wonderful
    ├── _\_init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Example online usage

```
cookiecutter https://github.com/audreyr/cookiecutter-pypackage.git
cookiecutter git+ssh://git@github.com/audreyr/cookiecutter-pypackage.git
cookiecutter hg+ssh://hg@bitbucket.org/audreyr/cookiecutter-pypackage
```

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 4 Exercises

Exercise 4-1 (cookies)

Create a new Django project called "cookies". Add an app named "oreo", with a **home** view that returns an HttpResponse with a text message. (E.g., "dunk me in milk")

Start the development server and go to http://localhost:8000/oreo to see your home page.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 5: Login for nothing and admin for free

Objectives

- Understand what the admin interface is for
- Implement admin for one or more apps
- Add and modify data via the admin interface

© 2023 CJ Associates *(Licensed to Protech)*

The admin Interface

- Admin for site users and groups
- Data entry and update for app data

Django provides an admin interface that allows you manage the users of your site (project). It also provides a simple interface to your models, which lets you perform CRUD functions from a browser.

To set up the admin interface, you need a superuser name and password, and you need to configure the admin interface for your apps.

The admin interface is intended for internal use; it it not designed to be a front end for your app.

© 2023 CJ Associates *(Licensed to Protech)*

Setting up the admin user

- Need an administrative user/password
- Use python manage.py createsuperuser
- Password must be >= 8 chars

Before you can set up the admin interface, you need to create the superuser – the admin for your site. This is done with the **createsuperuser** command to manage.py:

```
python manage.py createsuperuser
```

It will prompt you for name, email, and password.

```
$ python manage.py createsuperuser
Username (leave blank to use 'jstrick'): sitemgr
Email address: sitemgr@mysite.com
Password:
Password (again):
Superuser created successfully.
$
```

Configuring admin for your apps

- Register models with admin
- Usually in admin.py

To use the admin interface with your models, you need to register them. All that's needed is to edit the admin.py module in your app. Import the models you want to access with admin, and then call admin.site.register()

Example

djangoexamples/djmodels/superheroes/admin.py

```
from django.contrib import admin

# Register your models here.

from superheroes.models import Superhero, Power, Enemy, City # import your models

admin.site.register(Superhero)  # register each model
admin.site.register(City) # register each model
admin.site.register(Enemy) # register each model
admin.site.register(Power) # register each model
```


© 2023 CJ Associates *(Licensed to Protech)*

Using the admin interface

- Start development server
- Go to host:port/admin
- Log in as superuser

To use the admin interface, just start the development server and go to /admin. You'll need to log in with the superuser name and password, and then you will see your models listed. Click on any model to edit.

© 2023 CJ Associates *(Licensed to Protech)*

Tweaking the admin interface

- Create Admin model
- Add options and validations

By default, you can just register models with the admin interface, and they will work. However, for special cases you may want to create **Admin** models, which map to the actual models, and register the **Admin** models. These allow you to add options and validation to the admin interface for particular models.

To create an **Admin** model, define a class that inherits from admin.ModelAdmin.

Register that class by calling `admin.site.register()`. The first parameter is the model, the second is the model admin class.

© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/djmodels/superheroes/admin2.py

```
from django.contrib import admin

# Register your models here.

from superheroes.models import Superhero, Power, Enemy, City # import your models

admin.site.register(Superhero)  # register each model
admin.site.register(City) # register each model
admin.site.register(Enemy) # register each model
admin.site.register(Power) # register each model


class PowerAdmin(admin.ModelAdmin):  # create custom Admin model
    search_fields = ['name', 'description'] # add Admin metadata

admin.site.register(Power, PowerAdmin) # register custom model
```

See https://docs.djangoproject.com/en/1.10/ref/contrib/admin/ for more details on Admin modules

Table 2. ModelAdmin Options

Option	Description
`ModelAdmin.actions`	List of actions available on change list page
`ModelAdmin.actions_on_top`	Controls where actions bar appears
`ModelAdmin.actions_on_bottom`	Same…
`ModelAdmin.actions_selection_counter`	Whether selection counter displayed next to the action
`ModelAdmin.date_hierarchy`	Set date_hierarchy to name of DateField or DateTimeField
`ModelAdmin.empty_value_display`	Override default display value for empty fields
`ModelAdmin.exclude`	List of field names to exclude from form.
`ModelAdmin.fields`	List of fields to show
`ModelAdmin.fieldsets`	Set fieldsets to control layout of add/change pages
`ModelAdmin.filter_horizontal`	Use JavaScript “filter” interface for options
`ModelAdmin.filter_vertical`	Same as filter_horizontal, but vertical
`ModelAdmin.form`	Specify custom form for admin pages
`ModelAdmin.formfield_overrides`	Quick-and-dirty override of Field options
`ModelAdmin.inlines`	Add inline editing of related fields
`ModelAdmin.list_display`	Set which fields are displayed change list page
`ModelAdmin.list_display_links`	If and which fields in list_display linked to “change” page
`ModelAdmin.list_editable`	List of field names allowing editing on change list page
`ModelAdmin.list_filter`	Activate filters in right sidebar of change list page
`ModelAdmin.list_max_show_all`	Max items on “Show all” admin change list page
`ModelAdmin.list_per_page`	Max items on paginated list page
`ModelAdmin.list_select_related`	Use select_related() to retrieve objects on admin change list page
`ModelAdmin.ordering`	Specify how lists of objects should be ordered in admin views
`ModelAdmin.paginator`	Which paginator class is used for pagination
`ModelAdmin.prepopulated_fields`	Map field names to the values for prepopulation
`ModelAdmin.preserve_filters`	Preserve filters on list view
`ModelAdmin.radio_fields`	Use radio-buttons for foreign keys rather than select
`ModelAdmin.raw_id_fields`	List of static values for foreign keys
`ModelAdmin.readonly_fields`	List of fields to be non-editable
`ModelAdmin.save_as_continue`	Redirect to change view, otherwise changelist view

© 2023 CJ Associates *(Licensed to Protech)*

Option	Description
`ModelAdmin.save_on_top`	Add save buttons across top of admin change forms
`ModelAdmin.search_fields`	Enable search box on admin change list page
`ModelAdmin.show_full_result_count`	Control display of full count of objects
`ModelAdmin.view_on_site`	Whether or not to display “View on site”

Chapter 5 Exercises

Exercise 5-1 (music)

Set up the admin interface for the music project. Using the admin interface, add some more bands to your database.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 6: Creating models

Objectives

- Understand ORM
- Create and populate models
- Initialize the database
- Access data through the ORM

© 2023 CJ Associates *(Licensed to Protech)*

What is ORM?

- Object Relational Mapper
- Maps objects (Python classes) to DB tables
- No SQL needed
- Trades convenience for overhead

ORM stands for Object Relational Mapper. It refers to creating classes (in any language) that map to database tables. ORMs in other languages include Hibernate, NHibernate, and Spring.

Django has its own builtin ORM.

All data manipulation is done via the models; no SQL ever needs to be written. Behind the scenes, SQL is generated by Django routines. ORM adds a little overhead, but makes working with data very convenient.

For the rare situation that is not covered by ORM, Django provides a way to execute raw SQL.

The most popular Python ORM outside of Django is SQLAlchemy.

© 2023 CJ Associates *(Licensed to Protech)*

Defining models

- Python class corresponds to DBMS table
- Can specify constraints (foreign keys, etc.)
- Inherit from django.models

If you are starting your database from scratch, you will have to define models in terms of Django Model objects. These will ultimately be translated into SQL tables.

Every model needs a unique ID. By default, Django automatically adds an ID field which uses an auto-incrementing integer While ithis is fine for simple cases, it allows bad guys to guess record IDs. For better security, explicitly define an `ID` field that uses the `UUIDField` type, and populate it with a UUID from the `uuid` module.

For each "table", define a class that inherits from `django.db.models.Model`. Within each class, define a field (DB column) as a class variable, assigned from one of the Django model field types (see next page for list).

You can create foreign keys and other relations with special field types. The most common special field types are `ForeignKey` and `ManyToManyType`.

For each model, add a __str__() method. This should return a string that represents the model. Typically it will return the name field of the object, but it can return anything that makes sense to the developer, as long it it's a string.

© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/djmodels/superheroes/models.py

```
from django.db import models

class Power(models.Model):
    name = models.CharField(max_length=32, unique=True)
    description = models.CharField(max_length=512)

    def __str__(self):
        return self.name

class City(models.Model):
    name = models.CharField(max_length=32, unique=True)

    def __str__(self):
        return self.name

class Enemy(models.Model):
    name = models.CharField(max_length=32, unique=True)
    powers = models.ManyToManyField(Power)

    def __str__(self):
        return self.name

class Superhero(models.Model):
    name = models.CharField(max_length=32, unique=True)
    real_name = models.CharField(max_length=32)
    secret_identity = models.CharField(max_length=32)
    city = models.ForeignKey(City, on_delete=models.CASCADE)
    powers = models.ManyToManyField(Power)
    enemies = models.ManyToManyField(Enemy)

    def __str__(self):
        return self.name
```


© 2023 CJ Associates *(Licensed to Protech)*

Table 3. Django Model Field Types

Data fields	Relationship fields

`AutoField`
`BigAutoField`
`BigIntegerField`
`BinaryField`
`BooleanField`
`CharField`
`CommaSeparatedIntegerField`
`DateField`
`DateTimeField`
`DecimalField`
`DurationField`
`EmailField`
`FileField`
`FileField and FieldFile`
`FilePathField`
`FloatField`
`ImageField`
`IntegerField`
`GenericIPAddressField`
`NullBooleanField`
`PositiveIntegerField`
`PositiveSmallIntegerField`
`SlugField`
`SmallIntegerField`
`TextField`
`TimeField`
`URLField`
`UUIDField`
	`ForeignKey`
	`ManyToManyField`
	`OneToOneField`

© 2023 CJ Associates *(Licensed to Protech)*

Relationship fields

- Fields that relate to other models
- Three special field types
  - Foreign Key one-to-many
  - ManyToManyField many-to-many
  - OneToOneField one-to-one

The reason database systems are called relational is that tables (and in Django, models) are related to each other. There are several types of relationships.

A *foreign key relation* is one-to-many. An entry in one table contains the ID of an entry in another table, called the child table. Either end can be the "main" or "most important" table. A common example of this is customers and orders. In this scenario, each Order model has a Customer ID field that links it to a particular customer. In the Order model, Customer ID is a foreign key. In the Django ORM, you specify the name of the foreign model with a ForeignKey field.

A *many-to-many* relation occurs when you have two tables that might refer to each other. For instance, A book might have several authors, and an author might have several books. This is achieved in the database by having a third table that maps the association between the other two tables. The Django ORM creates this table automatically when you use a ManyToManyField. From one of the models, you specify the other side of the relationship in the ManyToManyField. Only do this on one side. That is, don’t use a ManyToManyField on both models. Either one is fine. The ORM will do the right thing.

A *one-to-one* relation is similar to a foreign key relation, except that there can only be one related object in the related model. It is implemented with the OneToOneField. Practically speaking, a model might have a list of values from its foreign key, but only one from a OneToOneField.

© 2023 CJ Associates *(Licensed to Protech)*

Creating and migrating models

- Managed by Django
- Keeps track of changes to database schema
- Allows reversion to earlier schema
- Like git for your database

Django will manage and keep track of changes to your database schema via *migrations*. Each update is numbered, and you can revert to previous versions.

A later chapter will cover data migration in detail.

To actually create the tables, run

```
python manage.py makemigrations
```

followed by

```
python manage.py migrate
```

This will execute the SQL to actually create the tables. Each time you change anything in your models, re-run the above commands.

Be sure the database (which can be empty) exists before running any Django commands

© 2023 CJ Associates *(Licensed to Protech)*

Using existing tables

- Django will build the models
- Use manage.py inspectdb > models.py
- Will model entire database – deleted unneeded tables

If you already have a database and your tables defined, you don't have to write the models yourself.

Use

```
python manage.py inspectdb
```

to generate the models for the entire database. This will generate models for all tables in the database, include system tables that you generally don't care about and don't want to update from your Django app.

Typically you would redirect the output to models.py, and then delete any models you didn't need. When you have narrowed models.py down to the models you actually need, then you may need to tweak and test the models until they work properly.

For instance, inspectdb sets all models to be unmanaged. This means that Django will not manage the model's lifecycle. An unmanaged model will be managed outside of your Django app. Change managed=False to managed=True in the Meta class within each model you want to have Django manage.

Once you have the models.py tweaked to your liking, use manage.py to run the makemigrations and migrate commands. Now you can access your existing tables without creating any models by hand.

TIP To use inspectdb with Oracle databases, the account needs schema owner privilege on at least dba_tables and dba_tab_cols.

© 2023 CJ Associates *(Licensed to Protech)*

Opening a Django shell

- Convenient for quick sanity checks
- Sets up Django environment
- Starts iPython (enhanced interpreter)

To interactively work with your models, you can open a shell. This opens a Python interpreter (nowadays iPython) with the project's configuration loaded.

This makes it easy to manipulate database objects.

To start the shell, type

```
python manage.py shell
```

Creating and modifying objects

- Create new object and populate
- Call save() on the object
- Equal to INSERT INTO or UPDATE

It is easy to create or update objects. Import the object from the models module, populate it as desired, and then call the save() method on the object.

You can create model instances using field names as named parameters, as well. Assigning invalid data will raise an error when save() is called.

To add foreign fields to an object, just create (or search for) the foreign object, and assign the foreign object to the appropriate field. Be sure to save the foreign object before you assign it.

To add many-to-many fields, use field.add(obj, ...).

© 2023 CJ Associates *(Licensed to Protech)*

Example

```
python manage.py shell

Python 3.6.2 |Anaconda custom (x86_64)| (default, Sep 21 2017, 18:29:43)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from superheroes.models import City, Enemy, Power, Superhero

In [2]: lex = Enemy(name='Lex Luthor')

In [3]: lex.save()

In [4]: met = City(name="Metropolis")

In [5]: met.save()

In [6]: flying = Power(name="flying", description="Fly though the air unaided")

In [7]: flying.save()

In [8]: sup = Superhero(name='Superman', secret_identity='Clark Kent', real_name='Kal-
el', ci
   ...: ty=met)

In [9]: sup.save()

In [10]: sup.powers.add(flying)

In [11]: sup.enemies.add(lex)

In [12]: sup.save()

In [13]: sup.secret_identity
Out[13]: 'Clark Kent'

In [14]: ^Z
```

Accessing data

- Import models from models.py
- Use MODEL.objects to access "rows"
- Use all(), filter(), or get() to select
- Use manage.py shell for interactive work

To access data in the database, you create a QuerySet object from your model, using a Manager object. The default Manager is called, confusingly enough, objects.

A QuerySet is equivalent to the result of a SELECT statement.

Use all() to retrieve all objects.

Use filter() to narrow down the results, like using WHERE in a SQL query. Filter takes named parameters that match the fields in the model.

Use get() to retrieve a single object by a particular field.

get() returns one object, but all() and filter() return QuerySets, which are lists of object.

© 2023 CJ Associates *(Licensed to Protech)*

Example

```
pm shell
Python 3.6.2 |Anaconda custom (x86_64)| (default, Sep 21 2017, 18:29:43)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from superheroes.models import City, Enemy, Power, Superhero

In [2]: h = Superhero.objects.all()

In [3]: for hero in h:
   ...:     print(hero.name, hero.secret_identity)
   ...:
Superman Clark Kent
Wonder Woman Diana Prince
Spider-Man Peter Parker
Iron Man Tony Stark

In [4]: spidey = Superhero.objects.filter(name='Spider-Man').first()

In [5]: spidey.real_name
Out[5]: 'Peter Parker'

In [6]: spidey.powers.all()
Out[6]: <QuerySet [<Power: super strength>, <Power: wallclimbing>, <Power: spider-
sense>]>

In [7]: spidey.enemies.all()
Out[7]: <QuerySet [<Enemy: Doctor Octopus>, <Enemy: Green Goblin>]>

In [8]: ww = Superhero.objects.get(name='Wonder Woman')

In [9]: ww.secret_identity
Out[9]: 'Diana Prince'

In [10]:
```

NOTE We will cover detailed manipulation of data in Chapter 8: Querying Django Models

Is ORM an anti-pattern?

- ORM not required for Django apps
- Use straight SQL, NoSQL, or flat files
- Views can get data from anywhere

One thing to remember is that just because you have a hammer, don't start treating everything like a nail.

ORMs provide a convenient mapping between language objects and database objects, but there is a lot of work in maintaining them. They can also be slower than straight SQL queries, especially when you get into multiple joins on very large tables.

There is nothing that requires you to use Django's ORM in your apps. You don't even have to use a database, although most apps do need to permanently store data, and a database is usually the most convenient way to do that.

However, there are a lot of factors. A web site could read a huge flat file into memory and provide data from an in-memory data structure. Maybe the file is updated every day, and the app re-reads it after each update.

All in all, ORM works pretty well, solves 80-90% of your database issues, and is well documented and well implemented.

Here are some links discussing ORM as an anti-pattern:

https://blog.codinghorror.com/object-relational-mapping-is-the-vietnam-of-computer-science/

http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

http://martinfowler.com/bliki/OrmHate.html

http://seldo.com/weblog/2011/08/11/orm_is_an_antipattern

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 6 Exercises

Exercise 6-1 (music)

Using the cookiecutter templates, create a new Django project named **music**, with an app named **bands**.

Define the models for the bands app. Populate the database with a few of your favorite bands.

Your models should include at least the following

```
Band
    Fields: name, genre, members
Genre:
    Fields: name
Member:
    Fields: name, bands
```

TIP In the Band model, members should be many-to-many, and genre should be a foreign key.

Chapter 7: Advanced Models

Objectives

- Recap model basics
- Implement str() and repr()
- Create custom managers and methods
- Configure metadata

© 2023 CJ Associates *(Licensed to Protech)*

Model recap

- Model == Table
- Class fields == Columns
- ID created automatically

To use a database in Django, you need to create **models**. Models are classes that inherit from django.models.Model. Columns (AKA Fields) are specified by class-level variables. They are the core of Django, and extremely flexible.

Every model needs to have a unique ID. If you do not specify a field as a primary key, Django will automatically add an integer ID field to the model.

For REST APIs, you should manually add an ID field of type **UUID**. This prevents malicious code (or people) from guessing record IDs by using IDs numerically close to known data. You can add the following line to all your models:

```
id = models.UUIDField(primary_key=True, default=uuid4, editable=False, help_text="Unique
ID of this <MODEL>")A
```

If you have an existing database, you can use **manage.py inspectdb** to define models from the tables. You will need to go in and tweak the models before use, however.

NOTE There may be issues when using inspectdb on Oracle databases.

Related Fields

- Many-to-one (AKA foreign key)
- Many-to-many
- One-to-one

To *normalize* your database means to avoid storing the same data in more than one place. For this to happen, tables can be related to each other.

There are at least three kinds of DB relations:

1. Many-to-one relations are called "foreign keys", and use the ForeignKey field type. This is used when a table has one value from another table. For instance, an Address model could have a foreign key to a City model.
2. Many-to-many relations are when two tables can relate to each other. For instance, a superhero can have multiple enemies, but an enemy could also have multiple superheroes. These require an intermediate table to connect them, which Django automatically builds.
3. One-to-one relations are less common, but are useful for "extending" tables. You might have a table representing a Person, with the usual fields, and a separate table contain bios, pictures, or other fields that are not used all the time. Keeping the tables separate makes queries on the "normal" field faster.

© 2023 CJ Associates *(Licensed to Protech)*

__str__() and __repr__()

- **__str__()** human-friendly
- **__repr__()** code to reproduce object
- **__repr__()** used if no **__str__()**

Whenever you create a model, you should define the **__str__()** and **__repr__()** methods.

The **__str__()** method should return a human-friendly version of the object. Thus, it usually uses one or more of the most significant columns. For a President object, the string might return "Abraham Lincoln" or "Teddy Roosevelt".

The **__repr__()** method should return a string containing, as nearly as possible, the code that would reproduce the object. For a President object, that might be "President(firstname="Abraham", lastname="Lincoln", ...)"

If you create a **__repr__()** but not a **__str__()**, the builtin **__str__()** from **object** will fall back to **__repr__()**.

```
class Wombat(models.Model):
    name = models.CharField(max_length=32)
    nickname = models.CharField(max_length=32)
    birth_city = models.ForeignKey(City, on_delete=models.CASCADE)

    def __str__(self):
        return self.name

    def __repr__(self):
        return "Wombat(name={0.name}, birth_city={0.birth_city})".format(self)
```

© 2023 CJ Associates *(Licensed to Protech)*

Fat, thin, and stupid

- Fat models
- Thin views
- Stupid templates

The authors of *Two Scoops of Django* claim to prefer "fat models", "thin views", and "stupid templates".

The idea is that most of the business logic for a model should be in the model itself. It shouldn’t be pushed out to the view, or even the template.

Views should also be thin. They should mostly just connect the right data to the right template.

Templates should be "stupid". They should not contain a lot of business logic; of course they contain presentation logic. Even here, tools like crispy forms can move much of that out to the models and forms themselves.

© 2023 CJ Associates *(Licensed to Protech)*

Null vs Blank

- In model
  - null=true
  - blank=true
- null is *missing* data in forms
- blank allows *empty* data in forms

A sometimes confusing aspect of Django models is the difference between using **null=true** and **blank=true** in model fields.

Setting **null** to true means that the field is not required in the database. If no value is specified when saving an object, NULL will be written to that field.

Setting **blank** to true means that it is OK to leave the field blank when filling out a form.

Just to confuse the issue, CharField and TextField send '' (empty string) to the database when empty, rather than null.

Default values

- Used if no value given
- Can be object or callable

When defining a module, you can specify a default value for each field. This can be either a normal value or a *callable* (function or callable class).

If it is a callable, it will be called each time an object is created.

© 2023 CJ Associates *(Licensed to Protech)*

Raw SQL calls

- *model*.objects.raw()
- Defer calls to unneeded fields

There are times when the ORM just won't do what you need. This might be some kind of complex JOIN statement, for instance. Or, you have a table with many columns, and you only need three of them.

You can call raw() from a manager with plain SQL code. It will execute the query on your database, and return a raw query set.

You can also import **db.connection** and get a database cursor, then use the cursor to execute a query and retrieve rows from the result set.

Custom Managers

- Add table-level functionality
- Aggregate functions
- Modify querysets

The **manager** is the object that a model uses to query the database. The default manager is named **objects**, as in Wombat.objects.all().

You can subclass models.Manager to build your own manager. In manager methods, **self** refers to the default manager, so you can query the model exactly like you would via *model*.objects.filter(), for instance.

You might do this to add table-level functionality to a model, such as grabbing data that needs more than one object, finding maximum and minimum values, etc.

Another reason to use a custom manager is for raw SQL queries. Moving raw queries out of views keeps the views more readable.

To use the manager, assign it to **objects** in the model.

```
class SuperheroManager(models.Manager):
    def get_fliers(self):
        return self.filter(powers__name__icontains="flight")

class Superhero(models.Model):
    ...
    objects = SuperheroManager()
    ...
```


© 2023 CJ Associates *(Licensed to Protech)*

Multiple model files

- make **models** folder
- add **__init__.py**
- put model modules in **models**

If a model file gets too large, you can split the models across multiple files. To do this, create a **models** folder in the same location as the **models.py** (usually the app dir). Put one or more .py files containing models in that folder.

To make this work, you *must* edit the **__init__.py** file and put the following:

```
from .modelfile import *
```

Do this for each separate models file.

NOTE You can also move custom managers into a file named managers.py (or whatever you like) and then, from the models files, import them as needed.

Model methods

- Add row-level functionality
- Define in models like normal methods
- **self** refers to the object

For "row-level" functionality you can add methods to any model class. The **self** parameter refers to an instance of the model (one object or database row). These methods can be called from the object in views.

This can be useful to manipulate data when it is hard to do using SQL, but you want to keep the logic in the model (where it belongs!), rather than writing more code in the view.

© 2023 CJ Associates *(Licensed to Protech)*

Model Metadata

- Define inner class Meta
- Add class-level data
- Controls non-field data

For adding more information to a model, you can define a class named **Meta** *inside* the model class. The Meta class can be used to provide extra information, including

- sorting order
- application label
- plurals for field names
- database table name
- whether Django manages the schema
- extra indexes on the model
- verbose names (singular and plural) for the model

Transactions

- Autocommit on by default
- Use ATOMIC_REQUESTS for per-request transactions

By default, Django uses autocommit mode, which means that whenever an object is saved, it is written to the database. Django uses transactions internally to protect the integrity of the database for any operations that require multiple DBMS operations.

In the database configuration, you can set ATOMIC_REQUESTS to true, which will cause each incoming request to be treated as a transaction. This is a simple and easy approach, but can lead to overhead issues in some cases.

You can also control atomicity on a view-by-view basis by using the transaction.atomic decorator. atomic() can also be used as a context manager (using the **with** statement).

```
from django.db import transaction

@transaction.atomic
def myview(request):
   pass

def myotherview(request):

    with transaction.atomic():
        # do DB operations
```


© 2023 CJ Associates *(Licensed to Protech)*

Chapter 7 Exercises

Exercise 7-1 (potusboot)

Use the potusboot project. Create a custom manager for presidents that has a new method **get_by_party()**. This will take the party as a string, and return a list of President objects for that party.

Create a view and template that will print the list of Whigs.

Add a model method **get_full_name()** that returns the full name of a president.

Use a Meta class to have the presidents sort by lastname, firstname. Make a view (and template) that displays a list of full names using your model method.

Chapter 8: Querying Django Models

Objectives

- Understand data managers
- Learn about QuerySets
- Use database filters
- Define field lookups
- Chain filters
- Get aggregate data

© 2023 CJ Associates *(Licensed to Protech)*

Object Queries

- Model class is table, instance is row
- model.objects is a manager
- QuerySet is a collection of objects

In the Django ORM, a model class represents a table, and a model instance represents a row. You have already seen some of this in the chapter on views.

To query your data, you will usually use a Manager. The default manager is named objects, and is available from the Model class.

From the object manager, you can get all rows, filter rows, or exclude rows. You can also implement relational operations, such as greater-than or less-than.

NOTE

For this chapter, there is just one view function that uses the builtin function eval() to evaluate a list of strings containing queries, in order to avoid duplicating the queries as labels.

Access this view at http://localhost:8000/superheroes/heroqueries

Example

djangoexamples/djsuper/superheroes/viewsqueries.py

© 2023 CJ Associates *(Licensed to Protech)*

```
from django.shortcuts import get_object_or_404, render
from django.db.models import Min, Max, Count, FloatField, Q
from .models import Superhero

q_hulk = Q(name__icontains="hulk")
q_woman = Q(name__icontains="woman")

def hero_queries(request):

    queries = [
        'Superhero.objects.all()',
        'Superhero.objects.filter(name="Superman")',
        'Superhero.objects.filter(name="Superman").first()',
        'Superhero.objects.filter(name="Spider-Man").first()',
        'Superhero.objects.filter(name="Spider-Man").first().secret_identity',
        'Superhero.objects.filter(name="Superman").first().enemies.all',
        'Superhero.objects.filter(name="Spider-Man").first().powers.all',
        'Superhero.objects.filter(name="Batman").first().powers.all',
        'Superhero.objects.exclude(name="Batman")',
        'Superhero.objects.order_by("name")',
        'Superhero.objects.count()',
        'Superhero.objects.aggregate(Count("name"))',
        'Superhero.objects.aggregate(Min("name"))',
        'Superhero.objects.aggregate(Max("name"))',
        'Superhero.objects.aggregate(Min("name"),Max("name"))',
        'Superhero.objects.filter(name__contains="man").count()',
        '''Superhero.objects.filter(
                name__contains="man").exclude(name__contains="woman")''',
        '''Superhero.objects.filter(
                name__contains="man").exclude(
                name__contains="woman").count()''',
        'Superhero.objects.all()[:3]',
        'Superhero.objects.filter(name__contains="man")[:2]',
        '''Superhero.objects.filter(
            enemies__name__icontains="Luthor").first().name''',
        'Superhero.objects.filter(q_hulk | q_woman)',
    ]

    query_pairs = [
        (query, eval(query)) for query in queries
    ]
    context = {
        'page_title': 'Query Examples',
        'query_pairs': query_pairs,
    }
    return render(request, 'superheroes/hero_queries.html', context)
```

Example

djangoexamples/djsuper/superheroes/templates/superheroes/hero_queries.html

```
{% extends "superheroes/superheroes_base.html" %}

{% block content %}
<h1>{{ page_title }}</h1>

<dl>
{% for query, result in query_pairs %}
    <dt>{{ query }}</dt>
    <dd>{{ result }}</dd>
    <br/>
{% endfor %}
</dl>

{% endblock %}
```


© 2023 CJ Associates *(Licensed to Protech)*

Opening a Django shell

- Convenient for quick sanity checks
- Sets up Django environment
- Starts iPython (enhanced interpreter)

To interactively work with your models, you can open a shell. This opens a Python interpreter (nowadays iPython) with the project's configuration loaded.

This makes it easy to manipulate database objects.

To start the shell, type

```
python manage.py shell
```

NOTE If iPython (recommended) is installed, the Django shell will use it instead of the builtin interpreter.

QuerySets

- Iterable collection of objects
- Roughly equivalent to "SELECT ..."
- Each row contains fields
- Use manager to create

A QuerySet is a collection of objects from a model. QuerySets can have any number of filters, which control which objects are in the result set. Filters correspond to the "WHERE ..." clause of a SQL query.

all(), filter(), exclude(), sortby(), and other functions return a QuerySet object. A QuerySet can itself be filtered, sorted, sliced, etc.

ORM queries are deferred (AKA lazy). The actual database query does not happen until the QuerySet is evaluated.

```
Superhero.objects.all()
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>, <Superhero: Wonder
Woman>, <Superhero: Hulk>]

Superhero.objects.filter(name="Superman")
[<Superhero: Superman>]

Superhero.objects.filter(name="Superman").first()
Superman

Superhero.objects.filter(name="Spiderman").first()
Spiderman

Superhero.objects.filter(name="Spiderman").first().secret_identity
Peter Parker

Superhero.objects.filter(name="Superman").first().enemies.all
[<Enemy: Lex Luthor>, <Enemy: General Zod>]

Superhero.objects.filter(name="Spiderman").first().powers.all
[<Power: Super strength>, <Power: Spidey-sense>, <Power: Intellect>]

Superhero.objects.filter(name="Batman").first().powers.all
[<Power: Detective ability>]

Superhero.objects.exclude(name="Batman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Wonder Woman>, <Superhero:
Hulk>]
```


© 2023 CJ Associates *(Licensed to Protech)*

```
Superhero.objects.order_by("name")
[<Superhero: Batman>, <Superhero: Hulk>, <Superhero: Spiderman>, <Superhero: Superman>,
<Superhero: Wonder Woman>]
```

Query Methods

Returning QuerySets
filter()
exclude()
annotate()
order_by()
reverse()
distinct()
values()
values_list()
dates()
datetimes()
none() (empty)
all()
union()
intersection() (1.11+ only)
difference() (1.11+ only)
select_related() (1.11+ only)
prefetch_related()
extra()
defer()
only()
using()
select_for_update()
raw()

Returning Objects
get()
create()
get_or_create()
update_or_create()
latest()
earliest()
first()
last()

Returning other values
bulk_create() (None)
count() (int)
in_bulk() (dict)
iterator() (iterator)
aggregate() (dict)
exists() (bool)
update() (int)
delete() (int)
as_manager() (Manager)

© 2023 CJ Associates *(Licensed to Protech)*

Field lookups

- Field comparisons
- Use *field*__operator
- Work with filter(), exclude(), distinct(), etc.

In SQL, the WHERE clause allows you to compare columns using relational operators. To do this Django, you can append special operator_suffixes to field names, such as name__greaterthan or secret_identity__contains. Note that the operators are preceded by two underscores.

Example

```
Superhero.objects.filter( name__contains="man").exclude(name__contains="woman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter( name__contains="man").exclude( name__contains="woman").count()
3
```

You can also create custom lookups for model fields

Field Lookup operator suffixes

Use in **filter()**, **exclude()**, and **get()**.

```
__exact
__iexact
__contains
__icontains
__in
__gt
__gte
__lt
__lte
__startswith
__istartswith
__endswith
__iendswith
__range
__date
__year
__month
__day
__weekday
__hour
__minute
__second
__isnull
__regex
__iregex
```


© 2023 CJ Associates *(Licensed to Protech)*

Aggregation Functions

- Calculate values over result set
- Use aggregate() plus calls to Count, Min, Max, etc
- Correspond to SQL COUNT(), MIN(), MAX(), etc

Some database tasks require calculations that use the entire result set (or subset). These are usually called aggregates. Common aggregation functions include count(), min(), and max().

To do this in Django, call the aggregate() function on a QuerySet, passing in one or more aggregation function. Each class is passed at least the name of the field to aggregate over.

aggregate() returns a dictionary where the keys are fieldname__aggregation_class, such as name__min. Parameters to aggregation functions may include a field name (positional), and the named parameter output_field, which specifies which field to return.

You can also generate aggregate values via the annotate() clause on a QuerySet

Aggregation Functions

```
Avg()
Count()
Min()
Max()
StdDev()
Sum()
Variance()
```

Example

```
Superhero.objects.aggregate(Count("name"))
{'name__count': 5}

Superhero.objects.aggregate(Min("name"))
{'name__min': 'Batman'}

Superhero.objects.aggregate(Max("name"))
{'name__max': 'Wonder Woman'}

Superhero.objects.aggregate(Min("name"),Max("name"))
{'name__max': 'Wonder Woman', 'name__min': 'Batman'}
```


© 2023 CJ Associates *(Licensed to Protech)*

Chaining filters

- Anything returning QuerySet can be chained
- Other methods are terminal (can't be chained)

Any QuerySet returned by some method can then have another method called on it; thus, you can chain filter methods to fine-tune what you need.

For instance, you could chain filter() and exclude(), then call count() on the result.

Example

```
Superhero.objects.filter(name__contains="man").count()
4

Superhero.objects.filter( name__contains="man").exclude(name__contains="woman")
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter( name__contains="man").exclude( name__contains="woman").count()
3
```

Slicing QuerySets

- Slice operator [start:stop:step] works on QuerySets
- Slice is lazy – only retrieves data for slice
- Use to fetch first N objects, etc.
- Negative indices are not supported

While a QuerySet is not an actual list, it can be sliced like most builtin sequences. Furthermore, the slice returns another QuerySet, so it doesn't execute the actual query until the data is accessed.

One difference from normal slicing is that negative indices and ranges are not supported.

Example

```
Superhero.objects.all()[:3]
[<Superhero: Superman>, <Superhero: Spiderman>, <Superhero: Batman>]

Superhero.objects.filter(name__contains="man")[:2]
[<Superhero: Superman>, <Superhero: Spiderman>]
```


© 2023 CJ Associates *(Licensed to Protech)*

Related fields

- Search models via related fields
- Use column__foreign_column

To search models using values in related fields, use column__foreign_column__lookup. This lets you apply field lookups to fields in the related column.

Example

```
Superhero.objects.filter(enemies__name__icontains="Luthor").first().name
Superman
```

Q objects

- Encapsulates SQL expression in Python objects
- Only needed for complex queries

By default, chained queries are AND-ed together. If you need OR conditions, you can us the Q object. A Q object encapsulates (but does not execute) a SQL expression. Q objects can be combined with the | (OR) or & (AND) operators. Arguments to the Q object are the same as for filters – field lookups.

Example

superheroes/queries/views.py

```
q_hulk = Q(name__icontains="hulk")
q_woman = Q(name__icontains="woman")
...
Superhero.objects.filter(q_hulk | q_woman)
[<Superhero: Wonder Woman>, <Superhero: Hulk>]
```


© 2023 CJ Associates *(Licensed to Protech)*

Chapter 8 Exercises

Exercise 8-1 (dogs/*)

In this exercise, you will start a project that you will use throughout the rest of the class.

- Create a project named **dogs** using cookiecutter-rest
- In the dogs project, create an app named dogs_core
- Configure the app
- Create models for Dog and Breed
  - Dog fields
    - name
    - breed (foreign key)
    - sex (m or f)
    - is_neutered
  - Breed fields
    - name
- Update the database
  - Create migrations
  - Migrate
- Add models to admin

Now use the Django shell to add some records for dogs and breeds. Add at least 6 dogs and at least 2 breeds. Be sure to add lots of variations so you can use them for searching.

Once you have created the records and saved them, use the query methods to find records as follows (vary the queries to match your own data):

1. All dogs
2. All breeds
3. Dog with specified name
4. All female dogs
5. All dogs of a selected breed
6. All female dogs whose name begin with 'B' *etc*

NOTE	As an optional enhancement, you could add a Category model, with categories such as working, herding, companion, sporting, etc. See https://www.akc.org/public-education/resources/general-tips-information/dog-breeds-sorted-groups/ for a list of which dogs

© 2023 CJ Associates *(Licensed to Protech)*

> belong in which categories. Category would be a foreign key to Breed.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 9: Migrating Django Data

Objectives

- Understanding migrations
- Using manage.py to migrate data
- Reverting (squashing) migrations

© 2023 CJ Associates *(Licensed to Protech)*

About migrations

- Changes to your database schema
- Speed up database changes
- Use scc on database schemas

After creating your initial database schema (table and column definitions), you will frequently need to add tables, as well as adding, deleting, and modifying columns.

Doing this manually is slow and error-prone.

Django 1.7 added migrations. A migration is a change to your data. Django tracks migrations through several utilities that can be called via manage.py. These tools make it much easier to propagate changes to your models into your database schema.

Migrations can be thought of as version control for your models.

© 2023 CJ Associates *(Licensed to Protech)*

Separating schema from data

- Schema is infrastructure (tables and columns)
- Data is ... *data*

While your database contains your data, it also contains tables and columns. This metadata is called the schema. When you modify a model, you will will be modifying your database schema when you migrate.

Django migration tools

- migrate
- makemigrations
- sqlmigrate
- showmigrations
- squashmigrations

The four commands provided for migration are shown above. They are called from manage.py.

migrate applies migrations, unapplies migrations, and lists status.

makemigrations checks for changes to your models and creates new migrations as needed.

sqlmigrate displays the SQL statements that will be executed for a migration.

showmigrations lists all of the migrations for a project.

squashmigrations combines multiple migratins in an efficient manner.

> **NOTE** migrations are applied project-wide by default.

© 2023 CJ Associates *(Licensed to Protech)*

Migration workflow

- Make changes to models
- Run manage.py makemigrations
- Run manage.py migrate

The normal workflow for working with migrations is:

1. Make changes to models as needed.
2. Run manage.py makemigrations This will create a set of migrations, which you can view with manage.py sqlmigrate.
3. Run manage.py migrate

This will apply any pending migrations.

NOTE Migrations are per-app, but run from the project perspective.

Adding non-nullable fields

- Add default value to model
- Add default value via `manage.py migrate`
- Migrate twice
  - Add field as nullable
  - Update values
  - Make field non-nullable

If you are adding a non-nullable field, the migrate command will need to know how to handle that field for existing records.

```
You are trying to add a non-nullable field '<new_field>' to <model> without a default; we
can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for
this column)
 2) Quit, and let me add a default in models.py
Select an option:
```

You can either add a default that will be applied to existing rows, or you can go back and add a default value to the field in the model.

If you need to add a field, but you can't add the same value to all existing fields, then you can migrate twice:

1. Create the field as nullable.
2. Create and run the first migration
3. Update the field as needed, making sure all rows have a value
4. Change the field to be non-nullable
5. Create and run the second migration

© 2023 CJ Associates *(Licensed to Protech)*

Migration files

- Normal python scripts
- Stored in the migrations folder

Migrations are implemented as normal Python scripts. They are stored in the migrations folder within the app.

Example

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [("migrations", "0001_initial")]

    operations = [
        migrations.DeleteModel("Tribble"),
        migrations.AddField("Author", "rating", models.IntegerField(default=0)),
    ]
```

Typical migration workflow

Typically, the workflow to update a production server might look like this:

LOCAL DEVELOPMENT ENV:

- Change your model locally
- Create new migrations (`manage.py makemigrations`)
- Migrate models (`manage.py migrate`)
- Test your changes locally
- Commit & push your changes to (git) server

ON THE PRODUCTION SERVER:

- Set ENV parameters
- Pull new code from **git**
- Update any new python dependencies (e.g. `pip install -r requirements.txt`)
- Migrate (`manage.py migrate`)
- Update static files (python manage.py collectstatic)
- Restart server

© 2023 CJ Associates *(Licensed to Protech)*

Squashing migrations

- Reduce set of migrations
- Does not remove old migrations
- Mostly automated

To cut down on the number of migrations needed, the **squashmigrations** command can be called from manage.py.

This will try to reduce all of the migrations up to a specified point into a smaller set of changes. It does not remove the original migrations, but when you run migrate commands, it will automatically used the squashed version.

Example

```
manage.py squashmigrations registry 0006
```

Reverting to previous migrations

- Select target migration
- Migrate to target
- Delete previous migrations
  - Use migration **zero** to unapply all

To revert to a previous migration, use the **migrate** command from **manage.py** with the latest migration to keep. In otherwords, if you have magration 0012, 0013, and 0014, and you want to revert to 0012, then say `manage.py migrate my_app 0012`.

To unapply all migrations, say `manage.py migrate my_app zero`.

© 2023 CJ Associates *(Licensed to Protech)*

Data Migrations

- Not automated
- Same as schema migrations
- Typically put in migrations folder

While Django does not have tools to automate data migrations, they are easy to create with the same tools that are used for schemas. You can write them as normal Python scripts and keep them in the migrations folder.

You have to follow the same format as schema migrations.

NOTE You could also add new commands to **manage.py**

Chapter 9 Exercises

Exercise 9-1 (dogs/dogs_core/models.py)

Add an an integer column "weight" to the Dog model. Use migration tools to update the database from the model.

HINT: Remember to do the migration in two steps (or use the option to provide a default).

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 10: Django Database Connections

Objectives

- Configure database information
- Work with multiple databases
- Migrate databases
- Choose databases in views

Database configuration

- DATABASES in settings
- Dictionary of connections
- Required: name, engine
- Optional: *many*

Databases are configured with the **DATABASES** setting. This is a dictionary with an entry for each connection. The key is the database alias for use within Django.

The alias "default" is special. It is used when a database is not specified. If you are only using one database, it is sufficient.

The value for each connection is a dictionary of per-connection settings. At least **name** and **engine** are required.

© 2023 CJ Associates *(Licensed to Protech)*

DATABASES options

Table 4. DATABASES options

Setting	Default	Description
`ATOMIC_REQUESTS`	`False`	Wrap each view in a transaction
`AUTOCOMMIT`	`True`	Use builtin transaction management.
`ENGINE`	`''`	Database backend to use
`HOST`	`''`	Database host (empty means localhost)
`NAME`	`''`	Database name. For SQLite, it's full path database
`CONN_MAX_AGE`	`0`	Connection lifetime in seconds. 0 to close connections per request; None for unlimited persistent connections.
`OPTIONS`	`{}`	Extra connection parameters
`PASSWORD`	`''`	Database password (not used with SQLite)
`PORT`	`''`	Database port (not used with SQLite)
`USER`	`''`	Database user name (not used with SQLite)
`TEST`	`{}`	Settings for test databases

Database backends

- Builtin
  - django.db.backends.postgresql_psycopg2
  - django.db.backends.mysql
  - django.db.backends.sqlite3
  - django.db.backends.oracle
- Available
  - MS SQL Server
  - SAP SQL Anywhere
  - IBM DB2
  - Microsoft SQL Server
  - Firebird
  - ODBC

Django ships with four backends (engines) for popular databases. Many backends are available for other databases.

The backend does the work of translating Django's generic models into the details for each database manager.

© 2023 CJ Associates *(Licensed to Protech)*

Multiple DBMS connections

- Load balancing
- Public/Private data
- Separate data server
- If "default" empty must specify db

Many projects use more than one database. This might be for load balancing, or for keeping private/internal data on a separate database from public data.

The databases do not have to be on the same machine, and they do not have to be the same type. You might use SQLite for Django's internal admin databases while using Oracle for actual site data.

A common scenario for multiple databases is load balancing. There could be one primary server, and 3 replicated servers. Updates would happen on the primary, but read-only access would happen on a randomly selected clone server.

Migrating multiple DBMS

- Only "default" migrated by default
- Specify with --database=*database*
- All models will be synchronized

When you run **python manage.py migrate**, only the default database will be migrated. Use the **--database** option to specify others.

All models will be synchronized to each database.

The **makemigrations** command is not per-database, but only looks for changes to the models. Migrations will be covered in detail in a later chapter.

NOTE Most database-related commands from manage.py use --database.

© 2023 CJ Associates *(Licensed to Protech)*

Selecting connection in views

- Default used if not specified
- *model*.objects.using("*dbalias*")
- *object*.save(using="*dbalias*")

When accessing data from multiple databases, querysets and **.save()** (or **.delete()**) will use the default database unless you specify another.

For general querying, the **.using** method is added to the normal manager (**objects**) to specify the database.

When saving or deleting, add the **using=** parameter.

Automatic routing

- Create router class
- Configure DATABASE_ROUTERS
- Use model name to decide

Most of the time you will be using either class-based views or viewsets. How can you specify which database to use? Plus, it's a bit of a nuisance to add the `.using('database')` method to all object manager calls.

The good news is you can automatically choose the database for a model by using *database routers*.

A database router is a class that intercepts read and write requests to the database. This is done at the project level, so that managers, class-based views, and viewsets all work as expected without additional configuration.

The router class can define any or all of the following methods:

Table 5. DB router methods

Method	Description
`db_for_read(model, **hints)`	called for any select queries
`db_for_write(model, **hints)`	called for any non-select queries
`allow_relation(obj1, obj2, **hints)`	called on joins
`allow_migrate(db, app_label, model_name=None, **hints)`	called during migration

In the db_* methods, check `model._meta.label` for *module.model_name*. Then return the alias (the label from the **DATABASES** setting) for the desired database. Put the router class in the config folder (or anywhere, as long as you specify the right path in settings).

Add the router class to the **DATABASE_ROUTERS** setting.

© 2023 CJ Associates *(Licensed to Protech)*

Example

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'presidents.db'),
    },
    # 'potus': {
    #     'ENGINE': 'django.db.backends.sqlite3',
    #     'NAME': '/put/path/to/DATA/presidents.db',
    # },
    'potus': {  # instructor only
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'scripts',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}

DATABASE_ROUTERS = [
    'presidents.dbrouters.PresidentRouter',
]
```

djangoexamples/presidents/presidents/dbrouters.py

```
class PresidentRouter:
    """
    Route president queries to the 'potus' database
    """
    def db_for_read(self, model, **hints):
        """
        Called for read operations on all databases

        :param model: model being queried
        :param hints: dict of hints; only key is "instance"
        :return: DB alias or None
        """
        if model._meta.label.split('.')[-1] == 'Presidents':
            return 'potus'

    # def db_for_write(self, model, **hints):
    #     pass
    #
    # def allow_relation(self, obj1, obj2, **hints):
    #     pass
    #
    # def allow_migrate(self, db, app_label, model_name=None, **hints):
    #     pass
```


© 2023 CJ Associates *(Licensed to Protech)*

Introspecting existing databases

- use `manage.py inspectdb`
- redirect to `models.py`

To use an existing database with Django, you still need models to make all the magic happen. There is no need to tediously construct models to match the existing database schema, however. Once the external database is configured in settings, use `manage.py inspectdb --database <database>` to dump all the tables from the database to standard output. You can redirect the output to models.py. If you only want to model a single table, use `manage.py inspectdb --database <database> <table>`.

```
python manage.py inspectdb --database <database> > <myapp>/models.py
```

You may need to modify the generated models. For the primary key field in the table, remove `null=True` nd add `primary_key=True`.

You can change the model name, but don't change the database table name or the datatypes of fields.

Once generated and modified, models can be used like any other Django models.

Accessing connection directly

- Use django.db.connections
- Dictionary-like object.
- Can get a cursor

To access a connection directly, import **connections** from **django.db**. This can be used for raw queries.

```
from django.db import connections
conn = connections['wombat2']

cur = conn.cursor()
cur.execute('select * from wombats')
results = cur.fetchall()
```


© 2023 CJ Associates *(Licensed to Protech)*

Chapter 10 Exercises

Exercise 10-1 (djangoexamples/aviation)

Create a project named **aviation** containing an app named **avcore** using cookiecutter. Migrate the admin databases as usual.

Configure a database with the alias "airports" in settings. It should point to the SQLite3 database **airports.db** in DATA. (Leave the default database).

Use manage.py to create models.py in **avcore**. It will create a model called **Airports**. Be sure to set the **code** field to be non-null, and mark it as the primary key.

Register the **Airports** model with the admin app (in admin.py).

Create a database router class named **AirportsRouter** in a module named **dbrouters.py** in the config folder. (aviation/aviation).

The router should redirect any read calls for the **Airports** model to the "airports" database.

Once everything is set up. Use either the admin app or the Django shell to query the database.

For the ambitious: make a simple page to display the airports.

Chapter 11: {chapter_title}

Objectives

- Learning REST guidelines
- Applying HTTP verbs to REST
- Aligning REST with CRUD
- Examining RESTful URLs
- Discussing searching and filtering

© 2023 CJ Associates *(Licensed to Protech)*

The REST API

- Based on HTTP verbs
  - GET get all objects or details on one
  - POST add a new object
  - PUT update an object (all fields)
  - PATCH update an object (some fields)
  - DEL delete an object

REST stands for *RE*presentational *S*tate *T*ransfer, first described (and named) by Roy Fielding in 2000. It is not a protocol or structure, but rather an architectural style resulting from a set of guidelines. It provides for loosely-coupled resource management over HTTP. REST does not enforce a particular implementation.

A RESTful site provides *resources*, which contain *records*. The same API typically contains more than one resource; each has a different *endpoint* (URL). For instance, `https://sandbox-api.brewerydb.com/v2/` has resources **beer**, **brewery**, and **ingredient**.

A RESTful API uses HTTP verbs to manipulate records. The same endpoint can be used for all access; what happens depends on a combination of which HTTP verb is used, plus whether there is more information on the URL.

If it is just the endpoint (e.g. *www.wombatlove.com/api/v1/wombats*): * GET retrieves a list of all resources. Query strings can be used to sort or filter the list. * POST adds a new resource

If it is the endpoint plus more information, typically a primary key (e.g. *www.wombatlove.com/api/v1/wombats/1*): * GET retrieves details for that resource * PUT updates the resource (replaces all fields) * PATCH updates the resource (replaces some fields) * DELETE removes the resource

NOTE see https://restfulapi.net/resource-naming/ for more information on designing a RESTful interface

© 2023 CJ Associates *(Licensed to Protech)*

REST constraints

There are six *constraints*, or guidelines, that make up REST. They are designed to be flexible. Remember, REST is an architecture, not a protocol.

Uniform Interface

- The interface defines the interactions between the client and the server (i.e., the client application and the RESTful API).
- Representations of resources contain enough information to alter or delete the resource.
- Data is sent via JSON (or maybe XML or HTML), rather than its "native" format.
- Every message has information that tells how to process the message.
- HATEOAS — all interaction is done via Hypermedia, using the URI, body contents, request headers, and parameters (Hypermedia as the Engine of Application State). While REST does not *require* HTTP, there are very few REST APIs that do not use it. A consequence of this is that the returned data should have links to retrieve the data requested or related data.

Stateless

- No sessions
- Requests contain all state (data) needed by server to fulfil the request. If multiple requests are needed, the client must resend, including authorization details.
- No client context stored on server between requests (client manages the state of the application)

Cacheable

- Responses must return uniform data (no timestamps, etc.) so it can be cached.
- Caching is important to reduce load on servers and infrastructure.

Client-Server

- Clients and server are completely independent
- Client only knows API (resource URIs)
- Server does not know (or care) how client uses data

Layered System

- APIs may be deployed on one server, data stored on another server, and validation on a third server, e.g.
- Client does not know (or care) about any servers other than the one providing the API

© 2023 CJ Associates *(Licensed to Protech)*

Code on Demand (optional)

- A REST-compliant API may optionally return executable code (e.g. GUI widget)
- This is unusual.

© 2023 CJ Associates *(Licensed to Protech)*

REST data

- JSON most popular format
- Any format is allowable
- Specified via request header

The data provided by a RESTful endpoint is usually in JSON format, but it doesn't have to be. The API can provide CSV, YAML, or other formats.

While some sites use a query string or infer from a resource suffix (i.e., `/wombats.csv`), the correct way to ask for a format is to speciy a MIME type in the request header.

```
response = requests.get('http://www.wombats.com/api/wombats', header={'accept',
'text/csv')
```

© 2023 CJ Associates *(Licensed to Protech)*

When is REST not REST?

- REST is guidelines, not protocol
- Implementers are not consistent
- YMMV (your mileage may vary)

REST is a set of guidelines, not a specific protocol. Because of this, REST implementations vary widely. For instance, many APIs use more than one endpoint for the same resource. Many APIs do not return a list of links on a GET request to the endpoint, but the details for every resource. Many APIs misuse (from the strict REST point of view) extended URLs.

For those sites, you have to read the docs carefully for each individual API to see exactly what they expect, and what they provide.

To avoid that, follow standard REST conventions, and your entire API should be easily discoverable by both humans and programs. Of course, your API will vary in the details of whatever data you're providing.

Public APIs

A list of public RESTful APIs is located here: http://www.programmableweb.com/category/all/apis?data_format=21190

You can use these to examine what they did wrong or, hopefully, did right.

© 2023 CJ Associates *(Licensed to Protech)*

REST + HTTP details

GET

A GET request can either get a list of resourses or the details for a particular resource.

With ID

If an ID is provided as part of the URI, a GET request should return the specified detail record.

Without ID

If given with no ID, the GET request should return all records for that resource, subject to query strings or default pagination. Pagination can be requested by the client or configured at the site or resource level. A successful resource request returns HTTP status 200 (OK)

Returns

200 for success, 4xx if no resource available

POST

The POST request creates a new record. It should be accompanied by JSON data in the body of the request.

Returns

201 (created) on sucess, 4xx for invalid data

PUT/PATCH

The PUT and PATCH requests update a new record. PUT replaces data in a record, and should provide values for all fields. PATCH updates a record and need only provide one or more values.

They should be accompanied by JSON data in the body of the request.

Returns

200 on success, 4xx for invalid data

DELETE

The DELETE request removes a specified record.

Returns

200 on success, 4xx for missing ID

© 2023 CJ Associates *(Licensed to Protech)*

Table 6. RESTful requests

Verb	URL	Description
GET	/API/wombat	Get list of all wombats
POST	/API/wombat	Create a new wombat
GET	/API/wombat/{id}	Get details of wombat by id
PUT	/API/wombat/{id}	Replace (all fields) wombat by id
PATCH	/API/wombat/{id}	Update (any fields) wombat by id
DELETE	/API/wombat/{id}	Delete wombat by "id"

© 2023 CJ Associates *(Licensed to Protech)*

REST best practices

Accept and provide JSON (not YAML, CSV, XML, etc.)

JSON is the standard. Use it.

Use nouns for endpoint paths (`/api/wombats`, not `/api/get_wombats`)

There is actually a lot of discussion on this point, but for most people, nouns sound most natural.

Use plural nouns (`/api/wombats`, not (`/api/wombat`)

See previous.

Use simple nesting that matches related resources (`/api/wombats/:id/sibling`), not (`/api/wombats/siblings?id=:id`)

While you can set up your URIs any way you like, structure them in a way that matches your related fields.

On error, return standard error codes

- 400 Bad Request — Client submitted incorrect data
- 401 Unauthorized — User is not authenticated
- 403 Forbidden — Authenticated user does not have permission for particular resource
- 404 Not found — Resource is not found (invalid ID, no results for query, etc)

Use secure IDs

Use UUIDs rather than sequential integers for ID fields, to prevent a hacker guessing ID numbers.

Use caching

Caching at any level will improve API throughput. Django has several builtin caching tools. For external tools you can use **Redis** and many others.

Keep versioning simple.

Some REST purists prefer to moving version information into the header. In this case, requests must put the key "Accept-version" in the request header, with a value such as "v1". This means that the URI will never change. However, it makes changing versions invisible to the client, who must know to put the right version in the header. In the absence of "Accept-version", sites usually return the latest version, which could break older client software.

© 2023 CJ Associates *(Licensed to Protech)*

The more common approach is to build the version into the URI:

```
https://www.wombats.com/api/v1/wombats
https://www.wombats.com/api/v2/wombats
```

and so forth. Keep the version part of the URL as simple as possible. It does not have to look like "v1", it can be any string — this will be handled in the URL config in each app, or in the project.

© 2023 CJ Associates *(Licensed to Protech)*

The OpenAPI Spec

- Originally called Swagger
- Describes site + resources
- YAML or JSON

The OpenAPI specification is a standard way of describing a RESTful API.

It grew out of a spec and toolset called **Swagger** originally created in 2010. The company **SmartBear** acquired the Swagger project in 2015, and donated the spec to the Linux foundation. The spec was renamed OpenAPI and is now open source. SmartBear provides commercial API tools under the Swagger brand.

The spec can be used to document an API before it is code, and (spoiler alert!) the Django REST framework can generate an OpenAPI spec from your models, serializers, and filters.

See the spec here: https://swagger.io/specification/

You can use the free API editor from Swagger to create or edit a spec.

https://swagger.io/tools/swagger-editor/

You can download the editor and install it using **npm, or use their online version. If you have trouble with the installation, then just open *index.html** with a browser.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 12: Django REST Framework Basics

Objectives

- Learn basics of the Django REST framework
- Define serializers for models
- RESTful routing
- Create function-based API views
- Create class-based API views

NOTE This chapter assumes a Django project is already created, with some available models.

About Django REST framework

- Flexible package for building REST APIs
- Includes OAuth authentication
- Supports ORM and non-ORM data
- Very customizable

The Django REST framework is a comprehensive package for creating RESTful APIs. It leverages the existing Django configuration and infrasructure.

The basic idea is to provide serializers that transform your models (or other data) into JSON (or possibly other formats) that can be returned via your application's REST API.

REST Framework provides class-based views, viewsets, and filters that make it easy to create REST apps.

A big feature of REST Framework is the browsable API.

REST Framework includes OAuth and other kinds of authentication, and supports both ORM and non-ORM data sources. It is extremely customizable.

© 2023 CJ Associates *(Licensed to Protech)*

Django REST Framework Architecture

- Serializers
- Filters
- Class-based views
- Viewsets

The crux of REST Framework, like a page-based Django app, is the model. REST Framework uses normal Django models. What it provides beyond that are serializers, filters, class-based views (CBVs), and viewsets. These work together to let you build apps with less coding.

For each model, you define a serializer. This is a class that converts the data in the object to native Python datatypes that can then be rendered into JSON (or some other format). In the serializer, you can control which fields are exposed to the API as well as performing conversions, etc.

Each model can also have a *filter*. This is a class that decribes query strings for the model. Filters allow you to fetch data with a URL like `⋯/api/contacts?name=Fred`.

To save writing a large number of similar view functions, REST Framework provides two kinds of classes that abstract away common tasks.

The first kind of classes are API views, which provides responses to HTTP requests such as GET or POST. They save the trouble of writing individual view functions for each different HTTP request. There are several basic class-based API views, plus mixins for customization.

The second kind are *viewsets*. Viewset further abstract the data from individual view functions. They work in conjunction with *router*, to automatically set up URL paths.

© 2023 CJ Associates *(Licensed to Protech)*

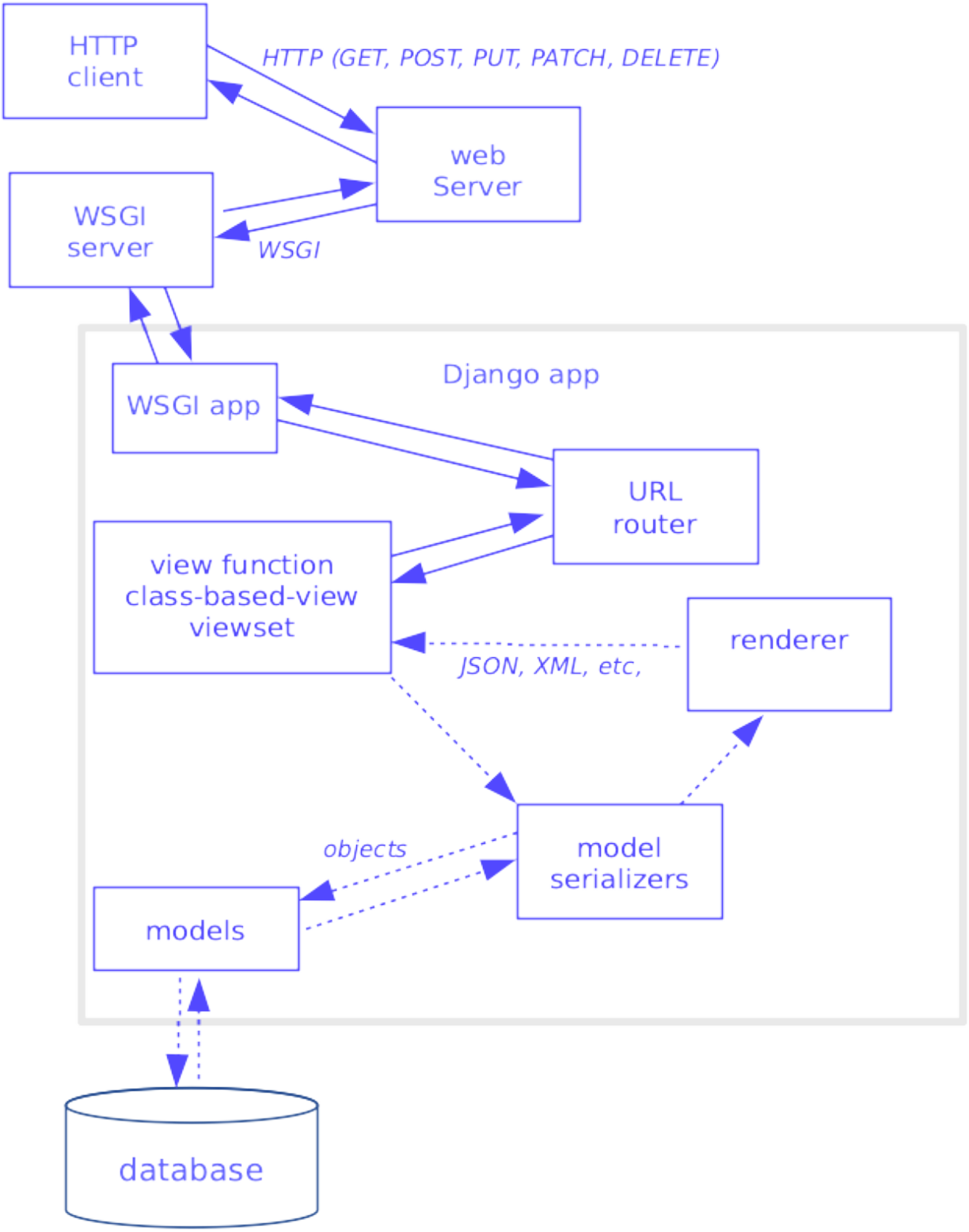


Figure 1. Django REST Architecture

© 2023 CJ Associates *(Licensed to Protech)*

Initial setup

- Add Django REST Framework to settings.py
- Create models
- Create serializers

To get started add **rest_framework** to INSTALLED_APPS in settings.

add a configuration dictionary to the project's settings.py file. This will contain all of the global settings for the REST API.

Serializing the hard way

- Use generic Serializer
- Specify all fields
- Define create and update methods

One way to create serializers is to start with a class that inherits from rest_framework.serializers.Serializer. Then add class variables that map to the fields in your models, or at least all the fields you want to expose in in your REST app.

Once this is done, the serializer class can be used with the matching model to serialize and deserialize the data.

To test serializers, they can be used in the Django shell:

```
In [20]: from contacts_core.models import City, Contact
In [21]: contact = Contact.objects.all().first()
In [22]: contact
Out[22]: <Contact: Contact object (99a5ba00-65bb-4ec0-8400-a7876fe6155c)>
In [23]: serializer = ContactSerializerPlain(contact)
In [24]: serializer.data
Out[24]: {'id': '99a5ba00-65bb-4ec0-8400-a7876fe6155c', 'first_name': 'John',
'last_name': 'Strickler', 'street_address': '4110 Talcott Dr.', 'postcode': '27705',
'dob': '1956-10-31', 'city': OrderedDict([('id', '7dc9cfb0-243f-46ea-b9a1-f1e29807308f'),
('name', 'Durham'), ('admindiv', 'NC'), ('country', 'US')])}
In [25]: from rest_framework.renderers import JSONRenderer
In [27]: json_data = JSONRenderer().render(serializer.data)
In [29]: json_data
Out[29]: b'{"id":"99a5ba00-65bb-4ec0-8400-
a7876fe6155c","first_name":"John","last_name":"Strickler","street_address":"4110 Talcott
Dr.","postcode":"27705","dob":"1956-10-31","city":{"id":"7dc9cfb0-243f-46ea-b9a1-
f1e29807308f","name":"Durham","admindiv":"NC","country":"US"}}'
```


© 2023 CJ Associates *(Licensed to Protech)*

Serializing the easier way

- Use ModelSerializer
- Reads metadata from models
- Only needs nested Meta class

The easy way to create serializers is to let the ModelSerializer class do all the work. It can read the metadata from your models and generate the serializer fields.

Example

djangoexamples/contacts/contacts_core/api/serializers.py

```
from rest_framework import serializers
from contacts_core.models import City, Contact


class AnimalSerializer(serializers.Serializer):

    animal = serializers.CharField(max_length=32)
    country = serializers.CharField(max_length=32)

class CitySerializerPlain(serializers.Serializer):

    id = serializers.UUIDField()
    name = serializers.CharField(max_length=32)
    admindiv = serializers.CharField(max_length=32)
    country = serializers.CharField(max_length=2)

class ContactSerializerPlain(serializers.Serializer):

    id = serializers.UUIDField()
    first_name = serializers.CharField(max_length=32)
    last_name = serializers.CharField(max_length=32)
    street_address = serializers.CharField(max_length=32)
    postcode = serializers.CharField(max_length=16)
    dob = serializers.DateField()
    city = CitySerializerPlain()

class CitySerializer(serializers.ModelSerializer):
    class Meta:
        model = City
        fields = ('id', 'name', 'admindiv', 'country')

class ContactSerializer(serializers.ModelSerializer):
    city = serializers.HyperlinkedRelatedField(view_name='contacts_core:api:city-detail',
read_only=True)

    class Meta:
        model = Contact
        fields = ('id', 'first_name', 'last_name', 'street_address', 'city', 'postcode',
'dob')
```


© 2023 CJ Associates *(Licensed to Protech)*

Implementing RESTful views

- Define JSON response from HttpResponse
- Import csrf_exempt decorator
- Create normal views
- Return JSON rather than HTML

For really simple cases, you can create more or less normal Django function-based views. However, you'll want to decorate them with the csrf_exempt decorator which protects them from cross-site scripting.

And of course the views should return JSON, not HTML, so use the serializers that were created earlier plus the JSON renderer.

Example

djangoexamples/contacts/contacts_core/api/fb_views.py

```
# not needed for REST CBVs or Viewsets
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework.renderers import JSONRenderer
from contacts_core.models import Contact
from .serializers import ContactSerializerPlain

# Create your RESTful views here.

# example without template (only used in class -- always use templates in real life):
@api_view(['GET'])
def hello(request):
     message = {"message": "Welcome to Contacts API Core"}
     renderer = JSONRenderer()
     return Response(renderer.render(message), 200)

@api_view(['GET'])
def contacts(request):
     contacts = Contact.objects.all()
     serializer = ContactSerializerPlain(contacts, many=True)
     contacts_json = JSONRenderer().render(serializer.data)
     return Response(contacts_json, 200)

@api_view(['GET'])
def contacts_detail(request, pk):
     contacts = Contact.objects.filter(id=pk)
```

```
    serializer = ContactSerializerPlain(contacts)
    contacts_json = JSONRenderer().render(serializer.data)
    return Response(contacts_json, 200)
```


© 2023 CJ Associates *(Licensed to Protech)*

Configuring RESTful routes

- Add views as normal to *app*/urls.py
- Allow for variable parts of URL
- Use named regular expression groups.

Add route configuration to the app's urls.py, and register them in the project's urls.py.

Example

djangoexamples/contacts/contacts/urls.py

```
from django.conf import settings
from django.urls import path, include
from django.contrib import admin

urlpatterns = [
    path('admin', admin.site.urls),
    # path('dogs', include('contacts_core.urls', namespace="dogs_core")),
    # path('art', include('contacts_core.urls', namespace="art_core")),
    path('', include('contacts_core.urls', namespace="contacts")),
    path('auth/', include('djoser.urls')),
    path('auth/', include('djoser.urls.authtoken')),
]

# include Django Debug toolbar if DEBUG is set
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__', include(debug_toolbar.urls)),
    ] + urlpatterns

# 3f2540e4b1bf996c396e04c5463a47e42900441a
```

Example

djangoexamples/contacts/contacts_core/urls.py

```
"""
URL Configuration for contacts_core
"""
from django.urls import path, include

app_name = 'contacts_core'

urlpatterns = [
    path('api/', include('contacts_core.api.urls', namespace="api")),
]
```


© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/contacts/contacts_core/api/urls.py

```
from django.urls import path, include
from rest_framework import routers

from . import fb_views
from . import cb_views
from . import viewsets

app_name = 'api'

router = routers.DefaultRouter()
router.register('contacts', viewsets.ContactViewSet)
router.register('cities', viewsets.CityViewSet)

urlpatterns = [
    # path('fbv/hello', fb_views.hello, name="hello"),
    path('fbv/hello', fb_views.hello, name="hello"),
    path('fbv/contacts', fb_views.contacts, name="fbcontacts"),
    path('fbv/contacts/<str:pk>', fb_views.contacts_detail, name="fbcontacts-detail"),
    path('cbv/contacts', cb_views.ContactsList.as_view(), name="contacts"),
    path('cbv/contacts/<str:pk>', cb_views.ContactsDetail.as_view(), name="cbcontacts-
detail"),
#    path('cbv/contactsx/<str:pk>', cb_views.ContactsList.as_view(), name="cbcontacts-
detail"),
    path('cbv/cities', cb_views.CitiesList.as_view(), name="cities"),
    path('cbv/cities/<str:pk>', cb_views.CitiesDetail.as_view(), name="cbcities-detail"),
    path('', include(router.urls)),
]
```

Class-based Views

- Builtin view functions
- Just need object and serializer
- Take care of rendering

While you *can* create function-based views, most developers use either viewsets or class-based views. We'll take a look at class-based views (CBVs) here, and viewsets get their own chapter.

To create a class-based view, import the app's models, and import classes from rest_framework.generics.

All that's needed for simple cases is to specify the queryset — the list of objects that the class represents, and the serializer class for those objects.

To use the class-based view, add it to a URL config. Since the URL config needs a *callable* (normally a function or method), CBVs have a method `as_view()` which returns a callable object that will implement the view.

Example URL config

```
path('cbv/contacts/<str:pk>', cb_views.ContactsDetail.as_view(), name="cbcontacts-
detail"),
```


© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/contacts/contacts_core/api/cb_views.py

```
# not needed for REST CBVs or Viewsets
from contacts_core.models import Contact, City
from rest_framework import generics
from  .serializers import ContactSerializer, CitySerializer

# class-based views (aka CBVs)
class ContactsList(generics.ListCreateAPIView):   #  GET /api/contacts  POST
/api/contacts

    queryset = Contact.objects.all()
    serializer_class = ContactSerializer

class ContactsDetail(generics.RetrieveUpdateDestroyAPIView):
    # GET /api/contacts/ID  PUT /api/contacts/ID  PATCH /api/contacts/ID  DELETE
/api/contacts/ID
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer

class CitiesList(generics.ListCreateAPIView):
    queryset = City.objects.all()
    serializer_class = CitySerializer

class CitiesDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = City.objects.all()
    serializer_class = CitySerializer
```

Chapter 12 Exercises

Exercise 12-1: (dogs/dogs_core/*)

Create a simple (non model-based) serializer for the Dog model, and a simple function-based view to display it.

Exercise 12-2: (dogs/dogs_core/*)

Create a model-based serializer for the Dog model, and a class-based view to display it.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 13: Django REST Viewsets

Objectives

- Understand viewsets
- Expose models using viewsets
- Provide pagination

What are viewsets?

- One viewset represents one resource
- Higher level of abstraction
- Tools for exposing models
- Less coding than class-based views

viewsets are REST Framework classes that go beyond class-based views to a higher level of abstraction. A viewset completely represents one resource.

They provide methods such as `list()` and `create()` rather than mapping directly to the HTTP verbs.

In practice, they mean that you have even less code to write. The combine all the logic for the views for a model in a single class.

Another advantage of viewsets is that they define their own routes, so you typically only have to add one line to your URL config for each model, and don't have to worry about naming the routes individually.

In terms of class-based views, a viewset would be a combination of `ListCreateAPIView` and `RetrieveUpdateDestroyAPIView`.

© 2023 CJ Associates *(Licensed to Protech)*

Creating Viewsets

- Import `viewsets`
- Define class
- Specify queryset and serializer

In many ways, using a viewset is similar to using a class-based view. You import a viewset, create a class to subclass it, and specify the model and serializer needed.

There are serveral base viewset classes. Which one to choose depends on how much customization there is. The most generic viewset is ViewSet, which requires you to write your own methods as needed.

The most commonly used viewset is ModelViewset, which exposes the data from a model, using the model's serializer.

NOTE you can customize viewsets in the same ways you can customize class-based views.

Example

djangoexamples/contacts/contacts_core/api/viewsets.py

```
from rest_framework import viewsets
from .serializers import ContactSerializer, CitySerializer
#  from contacts_core.api.filters import * # (optional)  change to only needed
serializers
from ..models import Contact, City

class ContactViewSet(viewsets.ModelViewSet):
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer
    # filter_backends = [DjangoFilterBackend]
    # filterset_class = MyFirstModelFilter
    # pagination_class = ...


class CityViewSet(viewsets.ModelViewSet):
    queryset = City.objects.all()
    serializer_class = CitySerializer
    # filter_backends = [DjangoFilterBackend]
    # filterset_class = MySecondModelFilter
```

Table 7. Available ViewSets

Viewset	Decription
`ViewSet`	Minimal viewset — add attributes (auth classes, permission classes) to define behavior
`GenericViewSet`	Inherits from GenericAPIView, but doesn't define actions
`ModelViewSet`	Includes implementations of standard actions.
`ReadOnlyModelViewSet`	like ModelViewSet, but read-only

© 2023 CJ Associates *(Licensed to Protech)*

Setting up routes

- Create a **router**
- Register viewsets with routers
- Add router.urls to URL config

To set up routes for a viewset, import **routers** from `rest_framework`.

Create a DefaultRouter (or other router) and register one or more viewsets with it. The name a viewset is registered with is incorporated into its route. These names are the resource endpoints.

In `urlpatterns`, use `include()` to delegate to the router-generated urls.

NOTE Routers, like everything else in Django, can be customized.

Example

djangoexamples/contacts/contacts_core/api/urls.py

```
from django.urls import path, include
from rest_framework import routers

from . import fb_views
from . import cb_views
from . import viewsets

app_name = 'api'

router = routers.DefaultRouter()
router.register('contacts', viewsets.ContactViewSet)
router.register('cities', viewsets.CityViewSet)

urlpatterns = [
    # path('fbv/hello', fb_views.hello, name="hello"),
    path('fbv/hello', fb_views.hello, name="hello"),
    path('fbv/contacts', fb_views.contacts, name="fbcontacts"),
    path('fbv/contacts/<str:pk>', fb_views.contacts_detail, name="fbcontacts-detail"),
    path('cbv/contacts', cb_views.ContactsList.as_view(), name="contacts"),
    path('cbv/contacts/<str:pk>', cb_views.ContactsDetail.as_view(), name="cbcontacts-
detail"),
#    path('cbv/contactsx/<str:pk>', cb_views.ContactsList.as_view(), name="cbcontacts-
detail"),
    path('cbv/cities', cb_views.CitiesList.as_view(), name="cities"),
    path('cbv/cities/<str:pk>', cb_views.CitiesDetail.as_view(), name="cbcities-detail"),
    path('', include(router.urls)),
]
```


© 2023 CJ Associates *(Licensed to Protech)*

Customizing viewsets

- Add class-level attributes
- Useful attributes
  - permission classes
  - lookup_field
  - pagination_class
  - filter_backends

In addition to `queryset` and `serializer_class`, there are many attributes than can be defined on a viewset.

`permission_classes` allows you to specify what permissions are required to access data in the view.

`lookup_field` specifies a field other than **pk** (the default) for item lookup.

`pagination_class` specifies a class for controlling pagination.

`filter_backends` provides filters as HTTP query strings to search the models.

See https://www.django-rest-framework.org/api-guide/generic-views/#genericapiview for a complete list of available attributes.

Adding pagination

- Controls number of items retrieved
- Can be set
  - App-wide
  - Per model

If your database has ten million **wombat** records and you make a generic request to list the **wombat** resource, it may overwhelm your client application. A well-designed API should allow limits and pagination.

REST Framework provides the **LimitOffsetPagination** class, which lets you provide a limit and an offset. The offset is the (0-based) first object to retrieve, and limit is the number of items to retrieve.

Also, using pagination adds `next` and `previous` fields to your result so that a client app can use them for navigation.

Normally, you want all views to have the same pagination, so you set it up in `settings.py` as one of the entries in REST_FRAMEWORK:

```
'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
```

Assign to the `pagination_class` attribute of a class-based view or viewset if you want to change the pagination scheme for that particular model.

NOTE set `pagination_class=None` to disable pagination.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 13 Exercises

Exercise 13-1 (dogs/dogs_core/*)

Add viewsets to your bands API.

(Optional): Add pagination.

Chapter 14: Django REST Filters

Objectives

- Design filters to enable query strings
- Create shortcuts for filters
- Use filters with viewsets or class-based views

© 2023 CJ Associates *(Licensed to Protech)*

What are filters?

- Automate querying of models
- Allow query strings on URLs
- For REST
  - Search records
  - Paginate (limit number of records)

Filters are classes that automate querying of models. Rather than having to write numerous `MyModel.objects.filter(⋯)` statement in hand-crafted viewset methods, you can specify which field you want to search and how you want to search them.

Filters can be used with both class-based views and viewsets.

For the details on creating filters, see https://django-filter.readthedocs.io/en/stable/guide/usage.html

Creating filters

- Import **rest_framework** from django_filters
  - Convenient to alias as **filters**
- Subclass **filters.FilterSet**
- Define filters

For each model create a *filteret* by subclassing **rest_framework.django_filters.FilterSet**.

In the filterset, add a filter for each field you want to filter. You can add more than one filter for a field. for instance, for a year field you could have **min_year**, **max_year**, and **year**. You can be creative — you could add **before_1950** as a filter.

Each filter is a filter type from **django_filters.rest_framework**. The name of the filter becomes the querystring on the URL for GET requests. Choose short, self-explanatory filter names, since these will be used by the clients.

Each filter must be listed in the **fields** attribute of the **Meta** class within the filterset. You also need to specify the model to be filtered in **Meta.model**.

For related queries, use `<related-field>__<relation>`.

© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/contacts/contacts_core/api/filters.py

```
from django_filters import rest_framework as filters
from contacts_core.models import *  # change to only required models

#  class MyFirstModelFilter(filters.FilterSet):
#      name = filters.CharFilter(field_name="name", lookup_expr='icontains')
#      birthplace = filters.CharFilter(field_name="place_of_birth",
lookup_expr='icontains')
#
#      """MAX BIRTH YEAR"""
#      min_birth_year = filters.NumberFilter(field_name="birth_year", lookup_expr='gte')
#
#
#      max_birth_year = filters.NumberFilter(field_name="birth_year", lookup_expr='lte')
#
#      min_death_year = filters.NumberFilter(field_name="death_year", lookup_expr='gte')
#      max_death_year = filters.NumberFilter(field_name="death_year", lookup_expr='lte')
#
#      alive = filters.BooleanFilter(field_name="death_year", lookup_expr='isnull')
#
#
#      class Meta:
#          model = MyFirstModel
#          fields = ["field1", "field2,]  # ...
#
#  class MySecondModelFilter(filters.FilterSet):
#      title = filters.CharFilter(field_name="title", lookup_expr='icontains')
#
#      class Meta:
#          model = MySecondModel
#          fields = ["field1", "field2",]  # ...
```

Filter shortcuts

- Auto-generates **equal** filter
- Use dictionary for flexibility
- Don't need to define individual filters.

If a field is listed in **Meta.fields**, but there is no filter defined, an "equal" filter will be created.

```
class ContactFilter(filters.FilterSet):
     class Meta:
         model = Contact
         fields = ["first_name", "last_name"]
```

If you want to use a different relation, **Meta.fields** can be a dictionary, where the key is the filter name, and the value is a list of relations:

```
class ContactFilter(filters.FilterSet):
    class Meta:
        model = Contact
        fields = {"first_name": ['icontains'], "last_name": ['icontains"]}
```


© 2023 CJ Associates *(Licensed to Protech)*

Using filters

- Specify in CBVs or viewsets
- Use query strings in URLs

To use filters in either class-based views or viewsets, add the following two fields:

filter_backend set to **django_filters.rest_framework.DjangoFilterBackend**

filterset_class set to the filterset class you defined.

Example

djangoexamples/contacts/contacts_core/api/viewsets.py

```
from rest_framework import viewsets
from .serializers import ContactSerializer, CitySerializer
#  from contacts_core.api.filters import * # (optional)  change to only needed
serializers
from ..models import Contact, City

class ContactViewSet(viewsets.ModelViewSet):
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer
    # filter_backends = [DjangoFilterBackend]
    # filterset_class = MyFirstModelFilter
    # pagination_class = ...


class CityViewSet(viewsets.ModelViewSet):
    queryset = City.objects.all()
    serializer_class = CitySerializer
    # filter_backends = [DjangoFilterBackend]
    # filterset_class = MySecondModelFilter
```

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 14 Exercises

Exercise 14-1 (labname.py)

Add filters to the **dogs** project. Clients should be able to filter on the **name**, **weight**, **breed**, and **sex** fields of **dog** objects, and on the **name** field of **breed** objects.

Run your project and try out the querystrings.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 15: Django REST Documentation

Objectives

- Generate browsable API docs
- Generate OpenAPI schemas

Documentation?

- Useful before coding
- Useful after coding
- Critical for client apps

Of course, it is important to provide documentation for applications. In the case of REST, it is crucial for anyone writing client code for your REST endpoints.

You can generate a schema from your existing API using various tools.

© 2023 CJ Associates *(Licensed to Protech)*

Generating an OpenAPI schema: Part I

- Use `pm generateschema
- Choice of formats

One way to generate a schema is to use the **generateschema** command that is part of **manage.py**. This generates a schema in YAML or JSON formats, with options to fine-tune the generation.

```
manage.py generateschema --format openapi --file myschema.yaml
```

Generating an OpenAPI schema: Part II

- Install **drf-spectactular**
- Add routes

To generate an OpenAPI schema, you can install the **drf-spectacular** extension to the REST Framework.

Then you just need to add routes to your URL config.

To install, use

```
pip install drf-spectacular
```

Add the following to **REST_FRAMEWORK** in `settings.py`.

```
'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
```

Then, in your project-level URL config, import views from `drf_spectacular.views`.

You can define route the three views as follows:

```
path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
# Optional UI:
path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='schema'),
name='swagger-ui'),
path('api/schema/redoc/', SpectacularRedocView.as_view(url_name='schema'), name='redoc'),
]
```

Now just visit the defined URLs.

- api/schema *download the schema as YAML*
- api/schema/swagger-ui *browsable Swagger/OpenAPI schema*
- api/schema/redoc *browsable ReDoc schema*

The ReDoc schema grew out of the Swagger schema, and provides a 3-panel layout.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 15 Exercises

Exercise 15-1 (dogs/dogs/*)

Use drf-spectacular to generate the OpenAPI spec for the dogs API.

Generate and test Python client code.

Chapter 16: Django REST Authentication

Objectives

- Implement token-based authorization
- Explore the **djoser** package
- Create client code

© 2023 CJ Associates *(Licensed to Protech)*

Authentication and authorization

- Validating users
- Permissions

First, some definitions:

Authentication is user validation, usually by mean of username and password. For security, only authenticated users can access a site.

Authorization is permission checking, at the project, model, or view level. Within the app, you can control which users have create, read, update, and delete permissions.

This can be managed either manually through the admin app, or programmatically through the API.

djoser

- **djoser** is auth extension for Django
- Provides token-based auth
- Optional
  - JSON Web tokens
  - 3rd-party (social media)

djoser is an extension for Django Rest Framework that provides an API-based auth system.

It add a set of views to your API.

It handles registration, login, logout, password reset, and account activation.

See https://djoser.readthedocs.io/en/latest/index.html for complete **djoser** docs.

© 2023 CJ Associates *(Licensed to Protech)*

Setting up djoser

- Add to INSTALLED_APPS
- Add routes

It is pretty simple to set up **djoser**.

Configure DEFAULT_AUTHENTICATION_CLASSES

```
REST_FRAMEWORK = {
  ...
    'DEFAULT_AUTHENTICATION_CLASSES': (
       'rest_framework.authentication.TokenAuthentication',
    ),
  ...
}
```

Add "djoser" to INSTALLED_APPS in your project settings. Make sure "rest_framework.authtoken" is also there.

```
INSTALLED_APPS = [
  ...
    'rest_framework.authtoken',
  ...
    'djoser',
  ...
]
```

Add routes for djoser in the project's URL config.

```
urlpatterns = [
  ...
    path('auth/', include('djoser.urls')),
    path('auth/', include('djoser.urls.authtoken')),
  ...
]
```

djoser endpoints

- token endpoints
- user endpoints
- for entire API

djoser provides a set of endpoints for authenticating and managing users. Auth settings and endpoints are for the entire project, even if you have multiple apps. However, you can fine-tune access by creating user or group permissions for a particular app.

- /auth/users/
- /auth/users/me/
- /auth/users/confirm/
- /auth/users/resend_activation/
- /auth/users/set_password/
- /auth/users/reset_password/
- /auth/users/reset_password_confirm/
- /auth/users/set_username/
- /auth/users/reset_username/
- /auth/users/reset_username_confirm/
- /auth/token/login/ (Token Based Authentication)
- /auth/token/logout/ (Token Based Authentication)
- /auth/jwt/create/ (JSON Web Token Authentication)
- /auth/jwt/refresh/ (JSON Web Token Authentication)
- /auth/jwt/verify/ (JSON Web Token Authentication)

NOTE In the above endpoints, "auth" is a configuration option in the URL config, but is commonly used. The host name is omitted, for clarity, so the entire url in real life might be `http://www.myapi.com/auth/users/`.

© 2023 CJ Associates *(Licensed to Protech)*

Registering a user

- POST to /auth/users
- Returns 201

To register a user, POST the username and password to `/auth/users/`.

© 2023 CJ Associates *(Licensed to Protech)*

Logging in/out

- POST to
  - /auth/token/login
  - /auth/token/logout
- login returns token
- logout returns ""

To log in, post username/password to `auth/token/login`. The JSON resone will return a temporary auth token via the key "auth_token".

This token must be provided with every request.

To log out, post to `auth/token/logout` with an empty payload, but with the auth header. This will return an tempty result.

© 2023 CJ Associates *(Licensed to Protech)*

Accessing the API

- GET/POST to api
- Provide token in auth header

To access the API, use the appropriate URL, such as `···/api/contacts/fc617ca1-623f-4a97-9d74-270cfbeafcef`.

You must send the auth_token via the "Authorization:" header entry with each request:

```
auth_token = ...   # get via login
headers = {'Authorization': f"Token {auth_token}"}

  response = requests.get(
      "http://127.0.0.1:8000/api/contacts",
      headers=headers
  )
  return response.content
```

Example

djangoexamples/contacts_access.py

```
import requests

USER_NAME = 'joeschmoe'
USER_PASSWORD = 'b00lab00la'

HOST = "http://127.0.0.1:8000/"

TOKEN_URL = f"{HOST}auth/token/"
LOGIN_URL = f"{TOKEN_URL}login"
LOGOUT_URL = f"{TOKEN_URL}logout"

USER_URL = f"{HOST}auth/users/"
PROFILE_URL = f"{USER_URL}me"

CONTACTS_URL = f"{HOST}api/contacts"

def main():
    result = register_user(USER_NAME, USER_PASSWORD)
    print("REGISTER:", result)

    profile = get_user_profile()
    print("PROFILE:", profile)

    auth_token = login(USER_NAME, USER_PASSWORD)
    print("AUTH_TOKEN:", auth_token)

    profile = get_user_profile(auth_token)
    print("PROFILE:", profile)

    contacts = get_contact_list(auth_token)
    print("CONTACTS:", contacts)

    result = logout(auth_token)
    print("LOGOUT:", result)

    contacts = get_contact_list(auth_token)
    print("CONTACTS:", contacts)

def register_user(username, password):
    response = requests.post(
        USER_URL,
        data = {
            'username': username,
```


© 2023 CJ Associates *(Licensed to Protech)*

```
            'password': password,
        }
    )
    return response.json()

def get_user_profile(auth_token=None):
    if auth_token:
        headers = {'Authorization': f"Token {auth_token}"}
    else:
        headers = {}
    response = requests.get(PROFILE_URL, headers=headers)
    return response.json()

def login(username, password):
    response = requests.post(
        LOGIN_URL,
        data={
            'username': username,
            'password': password,
        }

    )
    return response.json()['auth_token']

def logout(auth_token):
    headers = {'Authorization': f"Token {auth_token}"}

    response = requests.post(
        LOGOUT_URL, headers=headers
    )
    return response.content

def get_contact_list(auth_token):
    headers = {'Authorization': f"Token {auth_token}"}

    response = requests.get(
        CONTACTS_URL,
        headers=headers
    )
    return response.content

if __name__ == '__main__':
    main()
```

Chapter 16 Exercises

Exercise 16-1 (djangoexamples/dogs)

Implement token-based authentication to the **dogs** API.

Create some users and have them log in, get a token, and retrieve some dog info.

© 2023 CJ Associates *(Licensed to Protech)*

Chapter 17: Django Unit Testing

Objectives

- Design and implement unit tests
- Create urls using reverse()
- Test views via URLs
- Test RESTful APIs
- Use fixtures to simulate live data
- Run all tests

Django unit testing overview

- Tests go in tests/test*.py or tests.py
- Inherit from **django.test.testcase**
- Run via manage.py
- Use fixtures

By default, Django creates a tests.py module in every app. You can put your tests there, or you can put all tests for a project in a folder called tests. Conventional names are test_views.py, test_forms.py, etc.

Tests are defined in a *test case*. Test cases may be combined into a *test suite*.

Test classes inherit from django.test.TestCase, to handle some Django config issues.

NOTE If you want to put multiple test modules in the **tests** folder (i.e., package), you must import the tests in *tests/__init__.

© 2023 CJ Associates *(Licensed to Protech)*

Defining tests

- Inherit from django.test.TestCase
- Tests start with "test_"
- Each test asserts condition

django.test.TestCase inherits from the standard Python unittest.TestCase, adding Django-specific functionality. In particular, it provides an HTTP test client that has many useful features.

The basic idea is

Using the test client

- Make HTTP requests
- Follow redirects
- Confirm templates
- Check for content

The test client is part of the TestCase object.

To use it,

NOTE The Django test client can only retrieve pages that are part of your Django project. To retrieve pages from outside your Django project, use the **requests** module (or **urllib**).

© 2023 CJ Associates *(Licensed to Protech)*

Running tests

- Start at project or app folder
- Use **manage.py test**

To run tests, use

```
python manage.py test module_path
```

This will import the appropriate Django modules, and use the project's configuration information.

The module path can be just *modulename*, *modulename.TestCase*, or *modulename.TestCase.test_method*. If omitted, all tests in the project will be run.

About fixtures

- Sample ("mock") data
- Avoid using **real** database
- Created by manage.py dumpdata

To create static data for tests, you can create *fixtures*. These are datasets that can stand in for the real database during testing. This avoids using production or development databases which can change rapidly. If the database schema changes, new fixtures can be generated. You can create fixtures for any or all of a project's databases.

© 2023 CJ Associates *(Licensed to Protech)*

Creating fixtures

- Use manage.py dumpdata
- Redirect to a file
- Output is JSON by default

To create fixtures, use `manage.py dumpdata`. By default, this command will dump all of a project's database to JSON. The output goes to STDOUT, so it should be redirected to a file. The filename should end in ".json".

To only dump from a particular app or model, specify the app name or *app.model*. When specifying a single model, you can use the **pk** option to specify a list of record IDs (primary keys) to dump.

Use the **--format** option to change the output format to something other than JSON (i.e., YAML or XML). If you're just creating

Put the fixtures in a folder named **fixtures** within the app.

Example

```
pm  dumpdata --pks 1,3 superheroes.superhero
[{"model": "superheroes.superhero", "pk": 1, "fields": {"name": "Superman", "real_name":
"Kal-el", "date_of_death": "2017-12-04", "secret_identity": "Clark Kent", "city": 1,
"powers": [1, 2, 3, 4, 5], "enemies": [1, 2]}}, {"model": "superheroes.superhero", "pk":
3, "fields": {"name": "Spider-Man", "real_name": "Peter Parker", "date_of_death": "2017-
12-04", "secret_identity": "Peter Parker", "city": 3, "powers": [4, 7, 8], "enemies": [6,
7]}}](anaconda3-5.0.0) MacBook-Pro-4:djrest
```

Skipping tests

- Use @skip... decorators
  - Absolute
  - Conditional

unittest provides decorators for skipping tests.

`@unittest.skip("reason")` skips unconditionally. `@unittest.skipIf(condition, "reason")` skips if condition is true. `@unittest.skipUnless(condition, "reason")` skips if condition is false.

Why would you want to skip a unit test?

- A resource it needs is not available
- The test only runs on a particular platform

© 2023 CJ Associates *(Licensed to Protech)*

Reversing URLs

- Use `urls.reverse() for endpoints`
- Pass list of args

The first thing you might need is the URL of an endpoint to test. Rather than hard-coding the endpoint, use `django.urls.reverse()` to "reverse" the endpoing name into the actual URL.

```
reverse('contacts:api:contact-detail', args=[<obj_id>)
```

Testing REST APIs

- DRF provides test case
- Define class from test case
- self.client is API client

For testing APIs, the REST_Framework provides APITestCase. This is a verion of the standard Django test case that has additional features for testing APIs.

The general idea is to create tests which use `self.client()` to make API calls, and compare the results to what is expected.

You can import the models to get data directly from the database, and then make sure it matches what is retrieved.

You also should test user authorization for types access to resources.

© 2023 CJ Associates *(Licensed to Protech)*

Example

djangoexamples/contacts/contacts_core/tests/test_contacts_api.py

```
from rest_framework.test import APITestCase
from django.urls import reverse
from contacts_core.models import Contact, City

class TestContactsAPI(APITestCase):

    def setUp(self):
        self.invalid_id = "123abc"

        def contact_url(id):
            if id is None:
                args = []
            else:
                args = [id]
            return reverse('contacts:api:contact-detail', args=args)

        self.contact_url = contact_url

        def city_url(id):
            return reverse('contacts:api:city-detail', args=[id])

        self.city_url = city_url

    def test_retrieve_contacts_first_name(self):
        for obj in Contact.objects.all():
            expected = obj.first_name
            url = self.contact_url(obj.id)
            response = self.client.get(url)
            self.assertEqual(expected, response["first_name"])

    def test_invalid_contacts_id_returns_404(self):
        expected = 404
        url = self.contact_url(self.invalid_id)
        response = self.client.get(url)
        self.assertEqual(expected, response.status_code)

    def test_invalid_city_id_returns_404(self):
        expected = 404
        url = self.city_url(self.invalid_id)
        response = self.client.get(url)
        self.assertEqual(expected, response.status_code)

    def test_invalid_contact_id_returns_not_found(self):
```

```
        expected_key = "detail"
        expected_value = "Not found."
        url = self.contact_url(self.invalid_id)
        response = self.client.get(url)
        json_result = response.json()
        self.assertIn(expected_key, json_result)
        self.assertEqual(expected_value, json_result[expected_key])

    def test_retrieve_city_name(self):
        for obj in City.objects.all():
            expected = obj.name
            url = self.city_url(obj.id)
            response = self.client.get(url)
            self.assertEqual(expected, response["name"])
```


© 2023 CJ Associates *(Licensed to Protech)*

Chapter 17 Exercises

Exercise 17-1 (dogs/dogs_core/api/tests/*)

Add unit tests to your **dogs** project.

Run the unit tests and make sure they pass. Change your code and see whether they still pass.

```
python manage.py test
```

Appendix A: Django REST API Creation Checklist

- ▢ Create *project_name* using cookiecutter (or other layout creator)

```
cookiecutter ../SETUP/cookiecutter-rest
```

- ▢ Navigate to the *project_name* folder

```
cd project_name
```

- ▢ Run the first migration (optional for now)

```
python manage.py migrate
```

- ▢ Create *app_name* using cookiecutter (or other layout creator)

```
cookiecutter ../SETUP/cookiecutter-rest-app
```

- ▢ Add *app_name* to INSTALLED_APPS in *project_name.project_name*.settings.py
- ▢ Create one or more models in *project_name.app_name*.models.py
  - ▢ Add UUID as primary key to each model
- ▢ Create migrations and run them

```
python manage.py makemigrations
python manage.py migrate
```

- ▢ Register models in *project_name.app_name*.admin.py
- ▢ Create an app named **api** inside *app_name*
- ▢ Create one or more viewsets in *project_name.app_name*.api.viewsets.py
- ▢ Add routers and viewsets to the api's URL config in *project_name.app_name*.api.urls.py
- ▢ Include the URLs from api.urls to the app's URL config in *project_name.app_name*.urls.py
- ▢ Add the app's URL config to the project's URL config in *project_name.project_name*.urls.py
- ▢ From the project folder, start the development server

```
python manage.py runserver
```


© 2023 CJ Associates *(Licensed to Protech)*

Appendix B: Django Caching

About caching

- Web apps serve same pages over and over
- Each page needs work
- Caching skips redoing the work

A typical web app serves the same pages over and over, as users go to the same places and request the same data. Depending on the app, serving the page may involving fetching data from the database, performing some business logic on the data, and then passing the data to a template rendering function, which parses the template and fills in the data.

It is redundant to do this every time a user asks for a certain URL, so many web sites use caching, which stores returned pages and retrieves them directly without going through the view.

Caches may be set up to only cache for a certain amount of time, to only grow to a certain size, or both.

Python comes with **memcached,** which is robust and easy to set up. Other caching systems can be used, as well.

© 2023 CJ Associates *(Licensed to Protech)*

Types of caches

- In-memory
- Database
- File system
- Dummy

There are several locations to store cached pages. Many caches store pages in memory, although for huge sites this may not be feasible. For those sites, pages can be stored in a database or on the filesystem.

Django provides a dummy cache that does nothing. It is for use during development when you don’t yet care about caching. It is then easy to swap out the dummy backend for the real backend.

© 2023 CJ Associates *(Licensed to Protech)*

Setting up the cache

- Set CACHES in settings.py
- set up default cache unless using more than one
- Set BACKEND to Python package providing cache
- set LOCATION to IP/port or other as needed by cache
- Run manage.py createcachetable

To set up the cache, define the CACHES option in settings.py.

Unless you are using more than one cache system, you can use the default label.

Set BACKEND to the Python package that implements the cache, and set LOCATION to the connection information needed by the cache. This may be an IP/port or other data.

With most caching backends, LOCATION can be a list of multiple locations to spread the load over several machines.

Once caching is configured, run

```
manage.py createcachetable
```

To set up a table in the database to support caching.

Example

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```


© 2023 CJ Associates *(Licensed to Protech)*

Cache options

- CACHES takes many optional arguments
- Control size and timeout
- Key (data) manipulation

CACHES has many optional values to fine-tune how your pages are cached.

Example

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

Table 8. Cache Options

Option	Description
TIMEOUT	Default timeout, in seconds
OPTIONS	Options passed through tocache backend
MAX_ENTRIES	Maximum entries allowed
CULL_FREQUENCY	Fraction culled at MAX_ENTRIES
KEY_PREFIX	Prefix for all cache keys
VERSION	Version number for cache keys
KEY_FUNCTION	Path to a function composing key

© 2023 CJ Associates *(Licensed to Protech)*

Per-site and per-view caching

- Default is per-site
- Add apps to MIDDLEWARE in settings.py

To enable site-wide caching, add middleware apps to settings.py like this:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    ...
]
```

These apps most be in the order specified.

To enable caching only on selected views, use the cache_page() decorator from django.views.decorators.cache. This decorator takes one argument, an integer timeout in seconds.

While you can use this decorator in the modules that define your views (e.g.., views.py), this ties your app closely to the cache.

You can also wrap the decorator around views passed to url() in your URLconf:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```


© 2023 CJ Associates *(Licensed to Protech)*

Low-level API

- Access cache directly
- Store any picklable object (most builtin types)

django.core.cache.caches is a dictionary-like object that provides access to the caches. You can access keys via caches[key] lookup, or set key/value using caches.set(key, value).

Appendix C: Loading Django Data

Where do I start?

- Project database starts empty
- May need to add data for testing

Once you've created and migrated your models, your database is ready to use. And empty.

For a few records, you can add data via the admin interface or the Django shell.

However, you may want to add a number of records, for testing, or to migrate data from a previous incarnation of the project.

There are at least three ways to do this:

1. Create a data migration using the migration tools
2. Use **loaddata**
3. Create a new management command

NOTE	A fourth way is to write a script that is not part of your Django project, and that directly uses the database, but that is not recommended. It might create data that is incompatible with your app.

© 2023 CJ Associates *(Licensed to Protech)*

Creating a data migration

- Create an empty migration
- Implement code in migration file
- Call `manage.py migrate`
- Preferred approach

This is the preferred way to load data, as it becomes part of your data migration, so migrating on the deployment server is simpler.

To use migration tools, first create an empty migration with `manage.py makemigrations --empty <appname>`. This adds a migration file in the migrations folder, with no content. This will create a minimal migration script:

```
from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0004_auto_20201031_2202'),
    ]

    operations = [
    ]
```

The **dependencies** list is pre-populated with the previous dependency

To perform a data migration, create a function that takes two arguments, an app registry (typically called `apps`) that manages previous versions of the app, and a schema editor. For normal use, you won't need the schema editor.

The app registry will be used to fetch objects from the database that match the current migration. Use `apps.get_model(<appname>, <model>)`. In the function, perform whatever tasks are needed for your data migration. Remember to save all modified objects. You may want to write helper methods so that your main function doesn't get unwieldy.

Call your function with `migrations.RunPython(<function>)`.

```
from django.db import migrations

def copy_data(apps, schema_editor):
```

© 2023 CJ Associates *(Licensed to Protech)*

```
    Model1 = apps.get_model('myapp', 'Model1')
    Model2 = apps.get_model('myapp', 'Model2')
    for model1 in Model1.objects.all():
        model1.some_field = "some value"
        model1.save()

class Migration(migrations.Migration):


    dependencies = [
        ('myapp', '0004_auto_20201031_2202'),
    ]

    operations = [
        migrations.RunPython(copy_data)
    ]
```

NOTE

If you make a mistake and need to re-run your migration, use the `--fake` option to `manage.py migrate`. Use this option with the migration *before* the one you want to re-run. Then call `manage.py migrate` as usual and it will re-run the migration:

```
manage.py migrate --fake <previous migration>
manage.py migrate
```


© 2023 CJ Associates *(Licensed to Protech)*

Using loaddata

- Create JSON or YAML data file(s)
- Use `manage.py loaddata`

You can use the manage.py command **loaddata** to insert data into your database. The data must be in JSON or YAML format. To see the correct format, add at least one record using the Django shell or the admin interface, then call `manage.py dumpdata <APP.MODEL> <FILENAME>' for JSON, or `manage.py dumpdate --format yaml <APP.MODEL> <FILENAME>` for YAML.

If you have related fields, you will have to load one table to get IDs, then write code to use that table to build the next table, etc.

Creating a new management command

- Add command to manage.py
- Create commands in *app*/management/command
- Inherit from BaseCommand
  - Add arguments
  - Define **handle()**

If you need to do some specific tasks relative to your databases (or any part of your app), you can add new commands to manage.py for convenience. These commands have access to your project's Django configuration.

To add a new commmand, create a folder called `management` in an app, then create a subfolder called `commands`. In that folder, you can create any number of modules (scripts), each of which will be a separate command. For example, `validate_cities.py` could then be called as `manage.py validate_cities`.

In each command script, create a class named `Command` which subclasses `BaseCommand`. Add a class level variable named **help** which is set to a description of what the command does.

If the command needs arguments, define the method `add_arguments()`, which has a parameter `parser`. This parser object will parse arguments passed in after your command name when it is called via `manage.py`. Usage is

```
parser.add_argument('<argument name>', type=<argument type>)
```

You must define a method named `handler()`. This will be the code that runs when you call your command. It takes two arguments. `args` is not normally needed, and `options` contains the options parsed in `add_arguments()`.

If your command needs to work with the database, you can import your models at the top of the script.

© 2023 CJ Associates *(Licensed to Protech)*

Appendix D: Python Bibliography

Data Science

- ***Building machine learning systems with Python***. William Richert, Luis Pedro Coelho. Packt Publishing
- ***High Performance Python***. Mischa Gorlelick and Ian Ozsvald. O'Reilly Media
- ***Introduction to Machine Learning with Python***. Sarah Guido. O'Reilly & Assoc.
- ***iPython Interactive Computing and Visualization Cookbook***. Cyril Rossant. Packt Publishing
- ***Learning iPython for Interactive Computing and Visualization***. Cyril Rossant. Packt Publishing
- ***Learning Pandas***. Michael Heydt. Packt Publishing
- ***Learning scikit-learn: Machine Learning in Python***. Raúl Garreta, Guillermo Moncecchi. Packt Publishing
- ***Mastering Machine Learning with Scikit-learn***. Gavin Hackeling. Packt Publishing
- ***Matplotlib for Python Developers***.Sandro Tosi.Packt Publishing
- ***Numpy Beginner's Guide.Ivan Idris***.Packt Publishing
- ***Numpy Cookbook.Ivan Idris***.Packt Publishing
- ***Practical Data Science Cookbook.Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta***.Packt Publishing
- ***Python Text Processing with NLTK 2.0 Cookbook.Jacob Perkins***.Packt Publishing
- ***Scikit-learn cookbook.Trent Hauck***.Packt Publishing
- ***Python Data Visualization Cookbook.Igor Milovanovic***.Packt Publishing
- ***Python for Data Analysis.Wes McKinney.***. O'Reilly & Assoc

Design Patterns

- ***Design Patterns: Elements of Reusable Object-Oriented Software.Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides***.Addison-Wesley Professional
- ***Head First Design Patterns.Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra***.O'Reilly Media
- ***Learning Python Design Patterns.Gennadiy Zlobin***.Packt Publishing
- ***Mastering Python Design Patterns.Sakis Kasampalis***.Packt Publishing

General Python development

- ***Expert Python Programming.Tarek Ziadé***.Packt Publishing
- ***Fluent Python.Luciano Ramalho***. O'Reilly & Assoc.

© 2023 CJ Associates *(Licensed to Protech)*

- ***Learning Python, 2nd Ed..Mark Lutz, David Asher***. O'Reilly & Assoc.
- ***Mastering Object-oriented Python.Stephen F. Lott***.Packt Publishing
- ***Programming Python, 2nd Ed. .Mark Lutz***. O'Reilly & Assoc.
- ***Python 3 Object Oriented Programming.Dusty Phillips***.Packt Publishing
- ***Python Cookbook, 3rd. Ed.. David Beazley, Brian K. Jones***. O'Reilly & Assoc.
- ***Python Essential Reference, 4th. Ed..David M. Beazley***.Addison-Wesley Professional
- ***Python in a Nutshell.Alex Martelli***. O'Reilly & Assoc.
- ***Python Programming on Win32.Mark Hammond, Andy Robinson***. O'Reilly & Assoc.
- ***The Python Standard Library By Example.Doug Hellmann***.Addison-Wesley Professional

Misc

- ***Python Geospatial Development.Erik Westra***.Packt Publishing
- ***Python High Performance Programming.Gabriele Lanaro***.Packt Publishing

Networking

- ***Python Network Programming Cookbook.Dr. M. O. Faruque Sarker***.Packt Publishing
- ***Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers.T J O'Connor***.Syngress
- ***Web Scraping with Python.Ryan Mitchell***.O'Reilly & Assoc.

Testing

- ***Python Testing Cookbook.Greg L. Turnquist***.Packt Publishing
- ***Learning Python Testing.Daniel Arbuckle***.Packt Publishing
- ***Learning Selenium Testing Tools, 3rd Ed. .Raghavendra Prasad MG***.Packt Publishing

Web Development

- ***Building Web Applications with Flask.Italo Maia***.Packt Publishing
- ***Django 1.0 Website Development.Ayman Hourieh***.Packt Publishing
- ***Django 1.1 Testing and Development.Karen M. Tracey***.Packt Publishing
- ***Django By Example.Antonio Melé***.Packt Publishing
- ***Django Design Patterns and Best Practices.Arun Ravindran***.Packt Publishing
- ***Django Essentials.Samuel Dauzon***.Packt Publishing

© 2023 CJ Associates *(Licensed to Protech)*

- ***Django Project Blueprints.Asad Jibran Ahmed***.Packt Publishing
- ***Flask Blueprints.Joel Perras***.Packt Publishing
- ***Flask by Example.Gareth Dwyer***.Packt Publishing
- ***Flask Framework Cookbook.Shalabh Aggarwal***.Packt Publishing
- ***Flask Web Development.Miguel Grinberg***. O'Reilly & Assoc.
- ***Full Stack Python (e-book only).Matt Makai***.Gumroad (or free download)
- ***Full Stack Python Guide to Deployments (e-book only).Matt Makai***.Gumroad (or free download)
- ***High Performance Django.Peter Baumgartner, Yann Malet***.Lincoln Loop
- ***Instant Flask Web Development.Ron DuPlain***.Packt Publishing
- ***Learning Flask Framework.Matt Copperwaite, Charles O Leifer***.Packt Publishing
- ***Mastering Flask.Jack Stouffer***.Packt Publishing
- ***Two Scoops of Django: Best Practices for Django 1.11. Daniel Roy Greenfeld, Audrey Roy Greenfeld***.Two Scoops Press
- ***Web Development with Django Cookbook.Aidas Bendoraitis***.Packt Publishing

© 2023 CJ Associates *(Licensed to Protech)*

Index

© 2023 CJ Associates *(Licensed to Protech)*

© 2023 CJ Associates *(Licensed to Protech)*

T

U

V

W

© 2023 CJ Associates *(Licensed to Protech)*