Comprehensive Django Development

John Strickler

Version 1.0, March 2021

Table of Contents

About this Course

Welcome!

- We're glad you're here
- Class has hands-on labs for nearly every chapter
- Please write your name on the name tent if provided and not preprinted

Instructor name:

Instructor e-mail:

Classroom etiquette for 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.

Classroom etiquette for remote learning

- Please turn your mic off when you're not speaking. If multiple mics are on, it makes it difficult for all to hear
- The instructor doesn't know you need help unless you tell them. It's ok to ask for helpl 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

NOTE	Please do not bring killer rabbits to class. They might maim, dismember, or otherwise disturb your fellow students.

Course Schedule

Day 1

Chapter 1 Django Overview
Chapter 2 Getting started
Chapter 3 Using cookiecutter
Chapter 4 Creating models
Chapter 5 Login for nothing and the admin for free
Chapter 6 Creating Views

Day 2

Chapter 7 Basic Templates
Chapter 8 Querying models
Chapter 9 More about models
Chapter 10 Forms

Day 3

Chapter 11 Debugging and troubleshooting
Chapter 12 Class-based views
Chapter 13 Authentication
Chapter 14 Session management

Day 4

Chapter 15 Migrating Data
Chapter 16 Config
Chapter 17 About REST
Chapter 18 RESTful services
Chapter 19 Static file management

Day 5

Chapter 20 Unit Testing
Chapter 21 Caching
Chapter 22 Creating reusable apps
Graded Assessment

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 **py3djangocomp**.

What's in the files?

py3djangocomp contains data and other files needed for the exercises
py3djangocomp/EXAMPLES contains the examples from the course manuals.
py3djangocomp/ANSWERS contains sample answers to the labs.

The student files do not contain Django or Python itself. They will need to be installed separately.

NOTE	Installing Python, Django, and the student files may have already been done for you.

Extracting the student files

Windows

Open the file py3djangocomp.zip. Drag the expanded folder py3djangocomp to desktop. This will create the folder py3djangocomp.

NOTE	You can put the files anywhere you like, as long as you can find them again.

Unix (includes Unix, Linux, MacOS, etc)

Copy the file py3djangocomp.tgz to your home directory. Then, from your home directory, type

```
tar xzvf py3djangocomp.tgz
```

This will create the **py3djangocomp** directory, which contains all the student files.

Examples

- Most examples from course manual provided in EXAMPLES subdirectory
- First line of output shows how script was invoked

It will look like this:

Example

unicode.py

```
print(u"26\u00B0")
print(u"26\N{DEGREE SIGN}")
print(u"26\u00B0\n")
print(ur"26\u00B0\n")
print(ur"26\N{DEGREE SIGN}")
```

```
unicode.py
26°
26°
26°

26°\n
26\N{DEGREE SIGN}
```

Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in ANSWERS folder

Appendices

- Appendix A: Bibliography
- Appendix B: Packaging

Chapter 1: 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.

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.

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

Django in a nutshell

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

It's easy to get started with Django. The first step is to generate a project and an app.

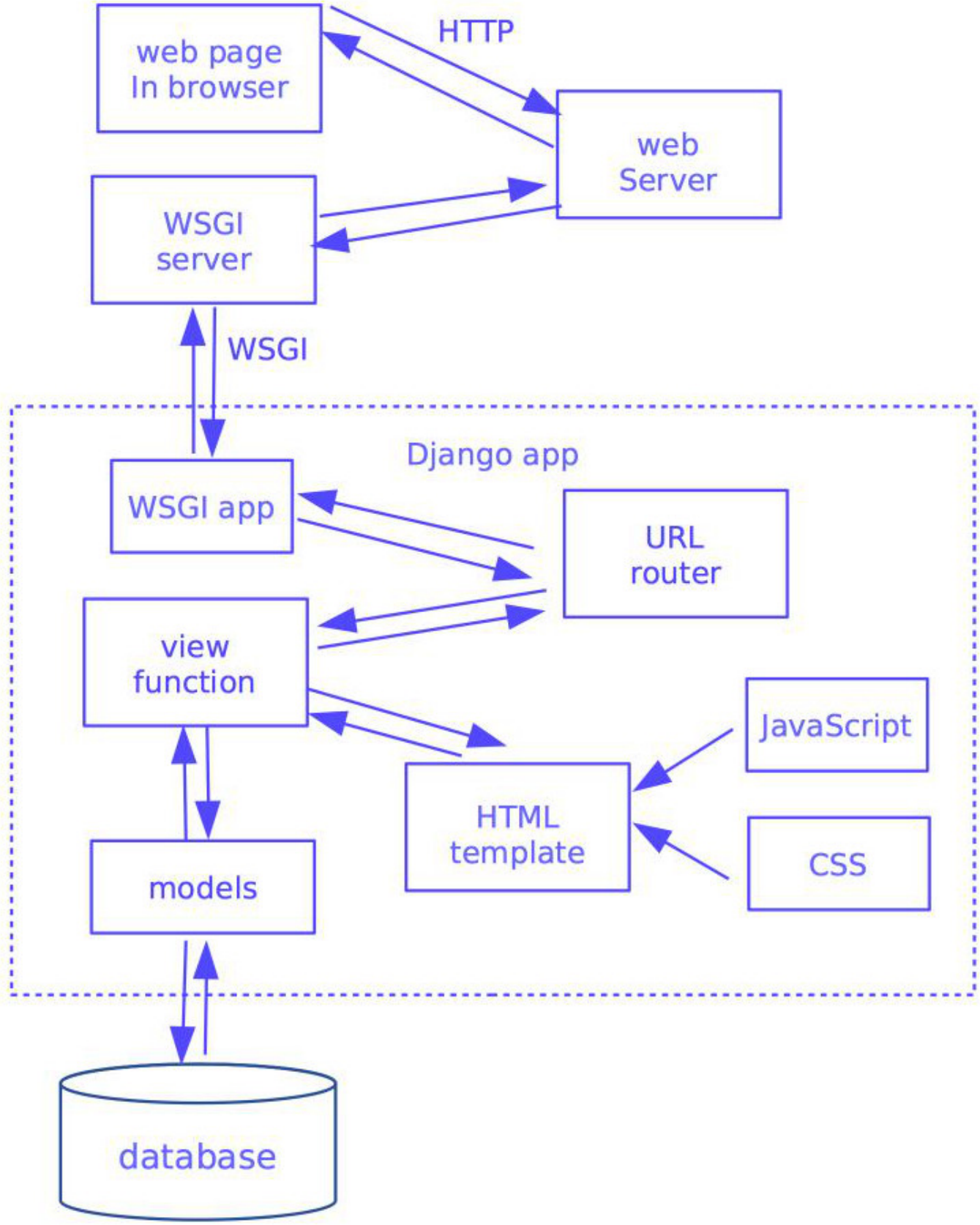
Assuming you will keep permanent data as part of the application, the next step is to define *models*. These map to tables in the database. The database can be new or existing. If it's existing, then Django can generate models from the database; if it's new, then you can define the models in Python and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

The next step is to create some *view functions*. A view function retrieves data via your models and passes the data to a template renderer, which returns the final web page that gets sent back to the browser.

Templates are files that contain the source to a web page – HTML, CSS, and JavaScript – as well as placeholders for app data. A view function sends data to a template renderer, which fills in the placeholders from the data and returns the final version of the web page.

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.

Django Architecture



Projects and apps

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

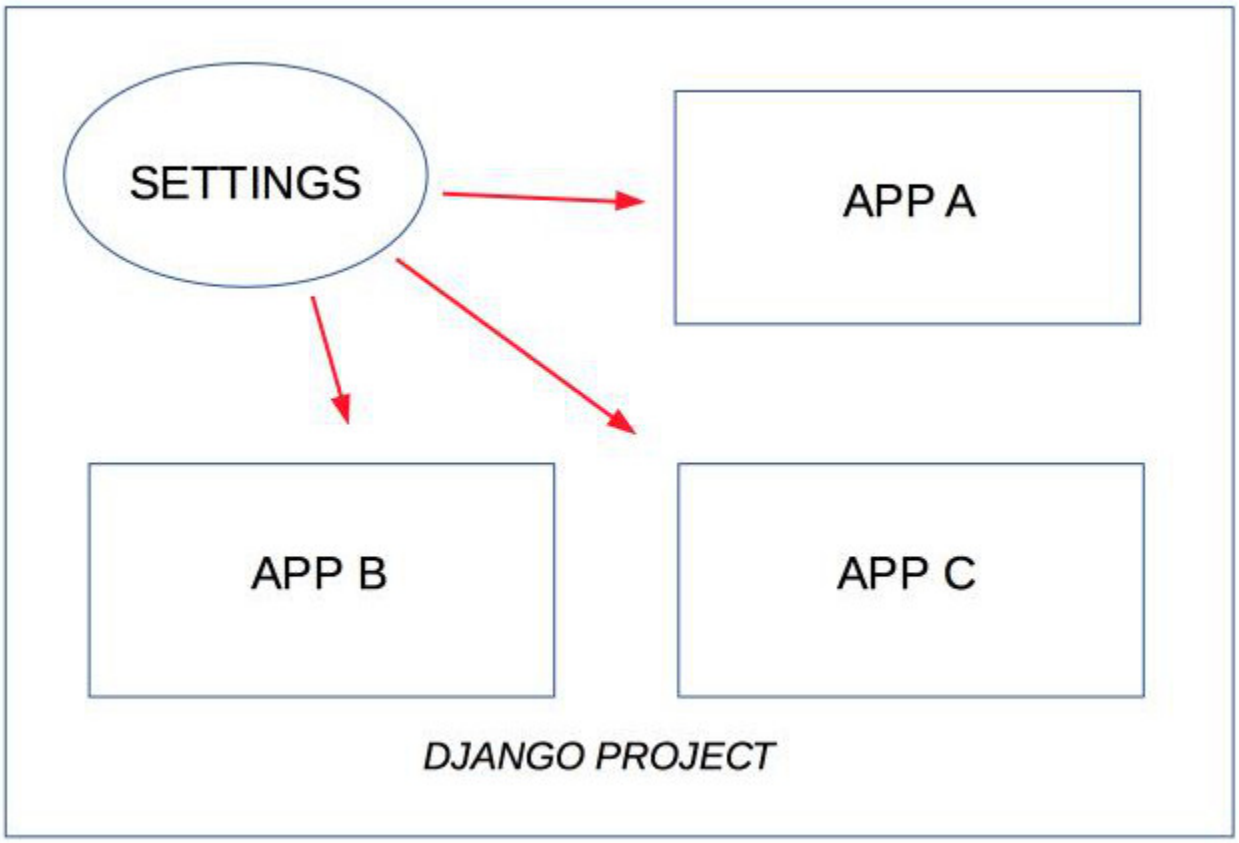
In Django, the first step is to create a project, and then create one or more apps within the project. While you can set up your project differently, it makes sense to work with Django's defaults.

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.



Chapter 2: Getting Started

Objectives

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

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.

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.

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

We'll look at **cookiecutter**, a more thorough 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.

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

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/1.9/ref/settings/

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.

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

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.

Add the app name (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 name to **LOCAL_APPS** in **config/base.py**.

Example

DJANGO/demo/demo/settings.py

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

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

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

DJANGO/demo/greet/views.py

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, Django World")
```

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

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.

Example

DJANGO/demo/demo/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:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/', include('blog.urls'))
"""
from django.conf.urls import url
from django.contrib import admin
from greet.views import home

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', home)
]
```

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.

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.

Chapter 2 Exercises

Exercise 2-1 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 3: Using Cookiecutter

Objectives

- Discover cookiecutter
- Use cookiecutter to start a project

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.

CAUTION	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.
```

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
```

Chapter 3 Exercises

Exercise 3-1 (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.

Chapter 4: Creating models

Objectives

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

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.

Defining models

- Python class <⇒ 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.

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.

Example

DJANGO/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
```

Table 2. Django Model Field Types

Data fields	Relationship fields
AutoField	ForeignKey
BigAutoField	ManyToManyField
BigIntegerField	OneToOneField
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	

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.

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

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.

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, ...).

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.

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

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

Chapter 4 Exercises

Exercise 4-1 (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 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

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.

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

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

DJANGO/djmodels/superheroes/admin.py

```
from django.contrib import admin

# Register your models here.

from superheroes.models import Superhero, Power, Enemy, City ①

admin.site.register(Superhero)  ②
admin.site.register(City) ②
admin.site.register(Enemy) ②
admin.site.register(Power) ②
```

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.

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.

Example

DJANGO/djmodels/superheroes/admin2.py

```
from django.contrib import admin

# Register your models here.

from superheroes.models import Superhero, Power, Enemy, City ①

admin.site.register(Superhero)  ②
admin.site.register(City) ②
admin.site.register(Enemy) ②
admin.site.register(Power) ②


class PowerAdmin(admin.ModelAdmin):  ③
    search_fields = ['name', 'description'] ④

admin.site.register(Power, PowerAdmin) ⑤
```

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

Table 3. 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

Option	Description
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
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.

Chapter 6: Creating Views

Objectives

- Learn what a view is for
- Translate MVC to MTV
- Create views
- Return 404 pages
- Use simple templates

Views

- Function returning content to browser
- Passed incoming HTTP request
- Usually return HTML
- The "controller" part of MVC

Views are functions that return content (usually HTML) to the client (usually a browser).

While a simple view might be self-contained, most views will pull data from your models and present it using a template.

If the requested data is not available, or some kind of error occurs, you will need to return an appropriate HTTP status code.

Views are passed the request object which has information from the client, including any data or parameters to be processed by the app.

Some of the data in the request object is parsed and made available separately.

NOTE	Despite being called "views", Django views are really the "controller" part of MVC as well as being some of the view logic.

Example

```
# very simple view
def home(request):
    return HTTPResponse("Welcome to my app")
```

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 includes the original full path of the request (the complete URL), the HTTP headers, and any cookies that are on the client's browser.

Table 4. HTTP request object attributes

Attribute	Description
scheme	Scheme of the request (usually http or https)
body	The raw HTTP request (byte string)
path	Full path to requested page (not including scheme or domain)
path_info	The path portion of the URL after the host name is split into a script prefix and path info
method	HTTP method (always uppercase)
encoding	Current encoding for form submission
content_type	MIME type of the request (parsed from the CONTENT_TYPE header)
content_params	A dictionary of key/value parameters from CONTENT_TYPE header
GET	Dictionary-like object with all HTTP GET parameters
POST	Dictionary-like object with all HTTP POST parameters
COOKIES	Dictionary containing all cookies (keys and values are strings)
FILES	Dictionary-like object containing all uploaded files
META	Dictionary containing all available HTTP headers
CONTENT_LENGTH	The length of the request body (as a string).
CONTENT_TYPE	The MIME type of the request body.
HTTP_ACCEPT	Acceptable content types for the response.
HTTP_ACCEPT_ENCODING	Acceptable encodings for the response.
HTTP_ACCEPT_LANGUAGE	Acceptable languages for the response.
HTTP_HOST	The HTTP Host header sent by the client.
HTTP_REFERER	The referring page, if any.
HTTP_USER_AGENT	The client’s user-agent string.
QUERY_STRING	The query string, as a single (unparsed) string.
REMOTE_ADDR	The IP address of the client.
REMOTE_HOST	The hostname of the client.
REMOTE_USER	The user authenticated by the Web server, if any.
REQUEST_METHOD	A string such as "GET" or "POST".
SERVER_NAME	The hostname of the server.
SERVER_PORT	The port of the server (as a string).

HttpRequest.resolver_match	Instance of ResolverMatch representing the resolved URL (only set after URL resolving takes place)

Route configuration

- Associates URL with view
- URL matched with regular expressions
- Two levels by default (but configurable)
  - *project*/*project*/urls.py
  - *project*/*app*/urls.py

To make views available in an app, they must be configured to match a particular URL. This is done with a URLconf.

When you create a project, it generates a default URL configuration is in project/project/urls.py. This initial config only includes the URL for the admin interface.

While you can do all the config in this file, it is usually more convenient to have app-specific config in the app. Thus, the project-level config includes a urls.py from each app.

URLs are added to a list named urlpatterns. Each pattern is a url object, which has at least two parameters. The first parameter is a regular expression that matches the URL, the second parameter is either a view function, an include(), or the as_view() method of a class-based view. django.conf.urls.include object, or a module containing a urlpatterns list.

You can add other named parameters that are passed into the view. This can be useful for customizing multiple URLs with the same view function, but makes the URL more dependent on the view.

In the project's **urls.py**, you can include URLconfs from apps. When doing this, specify a **namespace**. This can be used as a prefix when referring to URLs for that app. In the app's **urls.py**, you can include a name, which acts as a label for that URL. This makes it easy to move views around without hard-coding their paths in templates and view functions.

Example

DJANGO/djsuper/djsuper/urls.py

```
"""djsuper URL Mapping

The `urlpatterns` list maps URLs to views. More information:
    https://docs.djangoproject.com/en/1.11/topics/http/urls/

Function views:
    1. Add an import:  from my_app import views
    2. Add entry to urlpatterns:  url(r'^$', views.home, name='home')

Class-based views:
    1. Add an import:  from my_app.views import Home
    2. Add entry to urlpatterns:  url(r'^$', Home.as_view(), name='home')

Including another (usually an app's) URLconf:
    1. Import the include() function: from django.conf.urls import url, include
    2. Add a URL to urlpatterns:  url(r'^blog/', include('blog.urls', namespace="blog"))
"""
from django.conf import settings
from django.contrib import admin

# site-wide route mapping
from django.urls import path, include
urlpatterns = [
    path('admin', admin.site.urls),
    path('', include('superheroes.urls')),
]

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

Example

DJANGO/djsuper/superheroes/urls.py

```
"""
URL Configuration for superheroes
"""
from django.conf.urls import url
from . import views   # import views from app
from . import views404
from . import viewsbasictemplate
from . import viewstemplate
from . import viewsqueries

urlpatterns = [
    url('', views.home, name='home'),
    url('hero/<str:hero_name>/', views.hero, name="hero"),
    url('hero404x/<str:hero_name>/', views404.hero404, name="hero404"),
    url('hero404sc/<str:hero_name>/', views404.hero404sc, name="hero404sc"),
    url(
        'herotemplate101/<str:hero_name>/',
        viewsbasictemplate.hero_basic_template,
        name="herobasictemplate",
    ),
    url(
        'herohardway/<str:hero_name>/',
        viewstemplate.hero_hard_way,
        name="herohardway",
    ),
    url(
        'heroeasyway/<str:hero_name>/',
        viewstemplate.hero_easy_way,
        name="heroeasyway",
    ),
    url(
        'herolookups/<str:hero_name>/',
        viewstemplate.hero_lookups,
        name="herolookups",
    ),
    url(
        'herofilters/<str:hero_name>/',
        viewstemplate.hero_filters,
        name="herofilters",
    ),
    url(
        'herotags/<str:hero_name>/',
        viewstemplate.hero_tags,
```

```
        name="herotags",
    ),
    url(
        'herodetails/<str:hero_name>/',
        viewstemplate.hero_details,
        name="herodetails",
    ),
    url(
        'heroescape/<str:hero_name>/',
        viewstemplate.hero_escape,
        name="heroescape",
    ),
    url(
        'herourls/',
        viewstemplate.hero_urls,
        name="herourls",
    ),
    url(
        'herostatic/<str:hero_name>/',
        viewstemplate.hero_static,
        name="herostatic",
    ),
    url(
        'heroqueries/',
        viewsqueries.hero_queries,
        name="heroqueries",
    ),
]
```

RE Syntax Overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more branches separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of atoms. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a quantifier (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.

The following picture illustrates the the above concepts. S is a string anchor, A is an atom, W is a word anchor, and Q is a quantifier.

NOTE There is frequently only one branch.

Two good web apps for working with Python regular expressions are
https://regex101.com/#python
http://www.pythex.org/

Table 5. Regular Expression Metacharacters

Pattern	Description
.	any character
[abc]	any character in set
[^abc]	any character not in set
\w,\W	any word, non-word char
\d,\D	any digit, non-digit
\s,\S	any space, non-space char
^,$	beginning, end of string
\b	beginning or end of word
\	escape a special character
*,+,?	0 or more, 1 or more, 0 or 1
{m}	exactly m occurrences
{m,}	at least m occurrences
{m,n}	m through n occurrences
a|b	match a or b
(?aiLmsux)	Set the A, I, L, M, S, U, or X flag for the RE (see below).
(?:…)	Non-grouping version of regular parentheses.
(?P<name>…)	The substring matched by the group is accessible by name.
(?P=name)	Matches the text matched earlier by the group named name.
(?#…)	A comment; ignored.
(?=…)	Matches if … matches next, but doesn’t consume the string.
(?!…)	Matches if … doesn’t match next.
(?⇐…)	Matches if preceded by … (must be fixed length).
(?<!…)	Matches if not preceded by … (must be fixed length).

HttpResponse

- Standard response for views
- Specify HTTP status code
- No HTML needed (technically)

For simple web pages, you can use the HttpResponse object provided by Django. This object takes a string of text (which is typically, but doesn't have to be, HTML), and creates a standard HTTP response, including the default status code of 200 (OK).

Example

DJANGO/djsuper/superheroes/views.py

```
from django.http import HttpResponse
from .models import Superhero

def home(request):
    return HttpResponse("Welcome to the superhero app")

def hero(request, hero_name):
    s = Superhero.objects.get(name=hero_name)
    return HttpResponse(
        "{} is really {}".format(s.secret_identity, s.name)
    )
```

Extra URL information

- Part of URL after view name
- Passed into view as extra parameters
- Use named group

It is common to pass extra information to the view via the URL.

To do this, configure the route with a fixed part (the view) and a variable part. The variable part must be enclosed in a named group. The named group syntax is **(?P<_name_>_pattern_)**.

The group name will be the name of the parameter to the view function.

For example, the URL config:

DJANGO/djsuper/superheroes/urls.py

```
...
url(r'^hero/(?P<hero_name>.*)', views.hero, name="hero"),
...
```

Will pass everything after "hero/" into the view function as a parameter named "hero_name":

Example

DJANGO/djsuper/superheroes/views.py

```
from django.http import HttpResponse
from .models import Superhero

def home(request):
    return HttpResponse("Welcome to the superhero app")

def hero(request, hero_name):
    s = Superhero.objects.get(name=hero_name)
    return HttpResponse(
        "{} is really {}".format(s.secret_identity, s.name)
    )
```

Django 2.0 URL Configuration

- No REs needed
- Type is specified
- REs *may* be used

404 Errors

- Page (or data) not found
- Should raise Http404 exception
- Can display custom page
- Displays error page in debug mode

The Http404 class is an error type that will generate a standard 404 error page. You can create a template with the name **404.html**, and it will automagically be used.

get_object_or_404()

- Handles common use case
- Raise 404 error if record not found
- Pass Model, Manager, or QuerySet

A very common use case in web programming is to look up a record based on user input, which is generally a query string or a URL. If the record is found, then the app should display the data; otherwise, the app should display a 404 (not found) page.

Django provides a simple shortcut for this situation, get_object_or_404(). It takes a Model, Manager, or QuerySet, plus lookup parameters (similar to model.objects.get()). It performs the specified query. If the query fails, it raises a 404 error. Otherwise, you can use the object returned.

Example

DJANGO/djsuper/superheroes/views404.py

```
from django.http import HttpResponse
from django.shortcuts import Http404, get_object_or_404

from .models import Superhero

# Note: automagic 404 does not work when DEBUG=True

def hero404(request, hero_name):
    try:
        hero = Superhero.objects.get(name=hero_name)
    except:
        raise Http404("Hero '{}' not found [{}]".format(
                hero_name,
                request.get_full_path(),
            )
        )
    response = HttpResponse('<h1>Info for {} ({})</h1>'.format(
        hero_name,
        hero.secret_identity,
        )
    )
    return response


def hero404sc(request, hero_name):

    hero =  get_object_or_404(Superhero, name=hero_name)

    return HttpResponse('<h1>Info for {} ({})</h1>'.format(
            hero_name,
            hero.secret_identity,
        )
    )
```

About templates

- Separate presentation from data
- Get HTML, CSS, and JavaScript out of code
- Templates have placeholders for data

Even with shortcuts and writing to the HttpResponse object, there are still some issues with the views seen so far. We're still mixing the presentation (HTML, and potentially CSS and JavaScript) with the business logic. We want to be more like the MVC pattern (although Django calls it the MTV pattern – Model, Template, View).

To accomplish this, we can use templates. A template is a file containing presentation code (HTML, CSS, JavaScript), and placeholders for data. The view organizes the data from models, and passes the data and the template name to a renderer, which returns the final HTML page.

In addition to placeholders, the template can contain some directives, or simple business logic. The directives similar to, but simpler than Python itself, so they can be used by a web designer whose primary job is making the page look good, and not hard-core programming. Thus, the programmer creates the models and views, and the web designer creates the templates (whether or not they are the same person).

Using templates with views

- Get HTML, CSS, and JavaScript out of code
- Create templates folder in app
- Edit template and insert placeholders

As with all parts of Django, there is much flexibility in the use of templates.

The simplest way to use them is to go with Django's defaults. This assumes that the templates are located in a folder named templates in the app's folder.

The loader function in django.template will find and load the template, returning a template object. Then the template object's **render** method will fill out the template with the passed-in *context* dictionary and return the HTML ready to go.

Then pass the HTML to HTTPResponse as before.

NOTE In Chapter 7: Basic Templates you will see an easier way of finding, loading, and rendering the template.

Example

DJANGO/djsuper/superheroes/viewsbasictemplate.py

```
from django.http import HttpResponse
from django.shortcuts import get_object_or_404, render

from .models import Superhero


def hero_basic_template(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    context = {
        'hero_name': hero.name,
        'real_name': hero.real_name,
        'secret_identity': hero.secret_identity,
    }
    return render(request, 'hero_basic.html', context)
```

Template placeholders

- Delimited with {{ }}
- Refer to data name/value pairs passed into renderer

A template can be very complex, but the simplest template consists of normal HTML plus data placeholders of the form {{ NAME }}. When rendered, each placeholder will be replaced with the corresponding value for each NAME from a dictionary passed into the renderer. If the NAME is not in the dictionary, the placeholder is left as-is.

Example

DJANGO/djsuper/superheroes/templates/hero_basic.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero_name }}</title>
</head>
<body>
<h1>{{ hero_name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>
</body>
</html>
```

Chapter 6 Exercises

Exercise 6-1 (music/bands/*)

Create a view for your band app that takes a URL like bands/band/bandname and uses a simple template to present data for the specified band.

Chapter 7: Basic Templates

Objectives

- Learn mechanics of template use
- Serving templates "the hard way"
- Using the render() function
- Understand template variables
- Modifying variables with filters
- Implementing logic with tags
- Creating a consistent look and feel with inheritance

Django template overview

- `Text file with any content
- {{ variable }} {% tag %} {# comment #}
- Presentation logic
- Inheritance
- Builtin and custom filters

As noted in the previous chapter, it it important to separate business logic from presentation. In web programming terms, to separate the controller from the view.

The web page designer is concerned with HTML, CSS, and JS. The designer can create the web page without worrying where the data comes from. The template is created as a normal HTML page, plus directives from the Django Template Language, or DTL.

The Python programmer passes in a dictionary of variables (called the *context*) that the designer can use by placing variable placeholders in the template with {{ variable }}.

While business logic has no place in the view, it makes life easier to have some presentation logic. This can include loops, conditionals, and structural features such as nested templates. These are implemented as tags, which look like {{ if expression }} content {{ endif }}.

By design, the presentation logic does not resemble actual Python. The web designer does not need to learn an entire programming language, just a few DTL tags.

For fine-tuning variables, DTL provides filters, which are functions that can be applied to variables.

NOTE Django templates can be used for any kind of text. They are not specific to HTML

Example

```
<h1>{{ title }}</h1>
<ul>
{% for animal in animals %}
<li>{{ animal }}</li>
{% endfor %}
</ul>
{% if error_message %}
<span class="error">{{ error_message | upper }}</span>
{% endif %}
```

NOTE Django has builtin support for Jinja2, and you can add other template systems as well.

Default template configuration

- Default location app/templates
- load_template() for other locations
- Better: app/templates/app

Like every other component, Django has a very flexible configuration for templates. It starts in the settings file (site/settings.py) with the TEMPLATES setting.

The default setting works for many cases. The template shortcut functions will look for templates in a folder named **templates** in your app package. Thus, if you had an app named **spam** in a project named **ham**, and you specified a template named **eggs.html**, it will try to open **.../ham/spam/templates/eggs.html**.

Most developers like to put templates in a subfolder that matches the name of the app. While this seems redundant, it makes certain advanced features easier to implement. For the above example, you would specify spam/eggs.html as the template name, and it would be opened as .../ham/spam/templates/spam/eggs.html.

To load templates from other locations, update the TEMPLATES variable in settings.py.

This variable is a list of one or more dictionaries. Each dictionary configures one template processor, or "backend". The startproject command creates default settings for **DTL**, and a templates folder in the app package, as described above.

You can add paths to other locations to the DIRS list, and they will be searched in order. You can set the APP_DIRS to False, in which case the app/templates folder will not be searched. Otherwise, it is searched first.

Serving templates the hard way

- Find and load template
- Create context dictionary
- Render template
- Create HttpResponse from rendered template

While there are shortcuts for rendering templates, it’s good to know the "hard way".

The first step is to load a template. There are two functions to do this: **get_template()** and **select_template().**

get_template() finds a template and loads it. If the template can’t be found, it raises **TemplateDoesNotExist**. If it finds it, but the template has any errors, it raises **TemplateSyntaxError**.

select_template() is the same as get_template(), but takes a list of templates and returns the first template it finds. This is useful when you have specific templates based on content, but want to fall back to a more generic template if the first template is not found.

Both functions return a Template object.

Once you have a template, the next step is to create a context object. This contains the variables that will be used in the template.

Then you render the template, passing the context dictionary in to the **.render()** method, which returns the HTML page with the variables inserted into the placeholders.

Finally, return an HTTPResponse with the rendered HTML.

Example

DJANGO/djsuper/superheroes/viewstemplate.py

```
#!/usr/bin/env python
# (c) 2016 John Strickler
#
from django.shortcuts import get_object_or_404, render
from .models import Superhero
from django.http import HttpResponse
from django.template.loader import get_template

def hero_hard_way(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero_name': hero.name,
        'real_name': hero.real_name,
        'secret_identity': hero.secret_identity,
    }
    t = get_template('hero_basic.html')
    page = t.render(data)
    return HttpResponse(page)

def hero_easy_way(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero_name': hero.name,
        'real_name': hero.real_name,
        'secret_identity': hero.secret_identity,
    }
    return render(request, 'hero_basic.html', data)

def hero_lookups(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero,
    }
    return render(request, 'hero_lookups.html', data)

def hero_filters(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero,
    }
    return render(request, 'hero_filters.html', data)

def hero_tags(request, hero_name):
```

```
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero,
    }
    return render(request, 'hero_tags.html', data)

def hero_details(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero,
    }
    return render(request, 'hero_details.html', data)

def hero_escape(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero,
        'html_fragment': '<i>Some HTML</i>'
    }
    return render(request, 'hero_escape.html', data)

def hero_urls(request):
    context = {
        'title': 'Superheroes',
        'superheroes': Superhero.objects.all()
    }
    return render(request, 'hero_urls.html', context)

def hero_static(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero,
        'image_name': hero.name.lower().replace(' ', '_')
    }
    return render(request, 'hero_static.html', data)
```

DJANGO/djsuper/superheroes/templates/hero_basic.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero_name }}</title>
</head>
<body>
<h1>{{ hero_name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>
</body>
</html>
```

Serving templates the easy way

- Use the render() shortcut
- Parameters
  - request object
  - template name
  - dictionary of variables

You've already seen how to serve templates the hard way. The render() shortcut does the following:

- finds the template
- loads the template
- renders the template
- creates an HTTPResponse with the template

render() takes the three parameters. * the request object that was passed into the view function * the template name * a dictionary of variables, which will become the context for the template

Variable lookups

- {{ var }} scalar variable (int, float, str)
- {{ var.key }} dict key
- {{ var.attribute }} object attribute
- {{ var.method }} object method
- {{ var.index }} list or tuple index

In addition to using the value of a plain (scalar) variable, DTL supports elements of collections, and attributes of objects using "dot notation". They are searched in the order shown in the table: first dictionary keys, then object attributes, then object methods, then list or tuple indices.

Table 6. Django Template Language Dot Notation

Notation	Variable is	Returns
{{ var }}	Scalar (str, int, float)	Value of var
{{ var.key }}	Dictionary	Value of var["key"]
{{ var.attr }}	Object	Value of var.attr
{{ var.method }}	Object	Value of var.method()
{{ var.n }}	List or tuple	Value of var[n]

NOTE method must take no arguments

Example

DJANGO/djsuper/superheroes/templates/hero_lookups.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero.name }}</title>
</head>
<body>
<h1>{{ hero.name }}</h1>
<h2>Secret Identity: {{ hero.secret_identity }}</h2>
<h2>Real Name: {{ hero.real_name }}</h2>
<h2>First enemy: {{ hero.enemies.all.0 }}</h2>
<h3>Enemies list: {{ hero.enemies.all }}</h3>
</body>
</html>
```

Filters

- Modify variables
- Invoke with pipe symbol (|)
- var | filter same as filter(var) in Python
- Approximately 60 builtin filters

Filters are functions that can operate on DTL variables. The notation is var | filter, which is the equivalent of filter(var) in Python.

The idea is for the web designer to be able to modify the presentation without having to make a lot of micro adjustments to the web app Python source.

Filters can take one parameters. The syntax is var | filter:param2. This is equivalent to filter(var, param). If the parameter is a string, put quotes around it. (But not triple quotes).

There are no spaces allowed around the colon separating the filter from the parameter.

Example

DJANGO/djsuper/superheroes/templates/hero_filters.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero.name }}</title>
</head>
<body>
<h1>{{ hero.name }}</h1>
<h2>Secret Identity: {{ hero.secret_identity | upper }}</h2>
<h2>Secret Identity (short):
    {{ hero.secret_identity | slice:"0:5" }}
</h2>
<h2>Real Name: {{ hero.real_name | upper }}</h2>
<h2>First enemy: {{ hero.enemies.all.0 }}</h2>
<h3>Raw enemies list: {{ hero.enemies.all }}</h3>
<h3>Number of enemies: {{ hero.enemies.all | length }}</h3>
<h3>Enemies: {{ hero.enemies.all | join:", " }}</h3>
<h3>Enemies: <br/>{{ hero.enemies.all | join:"<br/>" }}</h3>

</body>
</html>
```

Comments

- {# commented text #} single line only
- {% comment "note" %} commented text – can be multiline {% endcomment %}

For short comments, enclose the comment in {# #}. For longer comments, use a comment block:

```
{% comment "label" %}  comment ... {% endcomment %}
```

Tags

- Presentation logic
- NOT Python
- Loops and conditionals
- Special features

Tags provide DTL presentation logic. In addition to flow control tags such as if, for, and while, there are many special-purpose tags.

Tags are not Python code, although the loops and conditionals look similar.

The {% if %} tag takes a variable, which might be passed through a filter, and, if it true, renders the template up to the {% endif %} tag. The {% for %} tag loops over an iterable. The loop variable can be used inside {{ }} like any template variable. Be sure to provide an {% endfor %} tag.

Example

DJANGO/djsuper/superheroes/templates/hero_tags.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero.name }}</title>
    <style>.blue { color: blue; } .red { color: red; }</style>
</head>
<body>
<h1>{{ hero.name }}</h1>

<h2>Secret Identity: {{ hero.secret_identity }}</h2>

<h2>Real Name:

{% if hero.real_name|length > 8 %}
    <span class="red">{{ hero.real_name }}</span>
{% else %}
    <span class="blue">{{ hero.real_name }}</span>
{% endif %}

</h2>

<h2>Enemies:</h2>
<ul>
{% for enemy in hero.enemies.all %}
    <h3><li>{{ enemy }}</li></h3>
{% endfor %}
</ul>
</body>
</html>

{% comment "just some notes" %}
This is some random text that
should not show up
in the page
{% endcomment %}
```

Inheritance

- Create base template for consistent look
- Keep CSS and JS includes in one place
- In parent: {% block blockname %} content {% endblock %}
- In child: {% extends parent_template %}
- In child: {% block blockname % } content { % endblock %}

You can create a base template that contains boilerplate HTML that you want to have on all your pages. In this base template, you can put one or more block tags. Then, you can create any number of child templates that define the same block tag, but contains child-specific content.

In the child template, use the {% extends templatename %} tag to inherit from a parent template. The extends tag must be the first tag in the template.

When you render the child template, it will use the base template, and replace the block with the same-named block from the child. This way you can have consistent headers/footers, etc.

By default, the contents of parent blocks will be overwritten when they are overwritten in the child. If you need to use the parent template content, use the builtin variable {{ block.super }} in the child.

Example

DJANGO/djsuper/superheroes/templates/superheroes_base.html

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <title>{% block title %}superheroes{% endblock title %}</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="">

    <!-- HTML5 shim, for IE6-8 support of HTML5 elements -->
    <!--[if lt IE 9]>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.3/html5shiv.min.js"></script>
    <![endif]-->

    {% block css %}
    <!-- Latest compiled and minified Bootstrap 4 beta CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/css/bootstrap.min.css" integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M" crossorigin=
"anonymous">

    <!-- Your stuff: Third-party CSS libraries go here -->

    {% endblock css %}

  </head>

  <body>

    <div class="container">

      {% block header %}{% endblock header %}
      {% block content %}
        <p>Default content for superheroes</p>
      {% endblock content %}
      {% block footer %}{% endblock footer %}

    </div> <!-- /container -->

    <!-- Javascript Placed at the end of the document so the pages load faster -->
```

```
    {% block javascript %}
      <!-- Required by Bootstrap v4 beta -->
      <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-
KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin=
"anonymous"></script>
      <script src
="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js" integrity
="sha384-b/U6ypiBEHpOf/4+1nzFpr53nxSS+GLCkfwBdFNTxtclqqenISfwAzpKaMNFNmj4" crossorigin
="anonymous"></script>
      <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/js/bootstrap.min.js" integrity="sha384-
h0AbiXch4ZDo7tp9hKZ4TsHbi047NrKGLO3SEJAg45jXxnGIfYzk4Si90RDIqNm1" crossorigin=
"anonymous"></script>

      <!-- Third-party javascript libraries go here -->

      <!-- App-specific Javascript in this file -->

      <script src="{% static 'js/superheroes.js'"></script>


    {% endblock javascript %}

  </body>
</html>
```

DJANGO/djsuper/superheroes/templates/hero_details.html

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

{% block content %}

<h1>{{ hero.name }}</h1>
<h2>Secret Identity: {{ hero.secret_identity }}</h2>
<h2>Real Name: {{ hero.real_name }}</h2>
<h3>Powers: {{ hero.powers.all | join:", "}}</h3>
<h3>Enemies: {{ hero.enemies.all | join:", " }}</h3>

<a href="{% url 'superheroes:herourls' %}">Return to hero list</a>
{% endblock %}
```

Escaping HTML

- HTML from variables auto-escaped by default
- Prevent XSS attacks
- Can turn off for desired raw HTML
- Many ways to do it
- safe filter
- {% auto-escape %} text {% endautoescape %}

By default, variables that contain any HTML are autoescaped. ">" is converted to ">", and so forth. Sometimes you want to be able to provide HTML (or CSS or JS) in a template variable. To turn autoescaping off, use the safe filter. This tells the renderer that the text is "safe".

The HTML will be passed through unchanged.

Autoescape can also be turned off in the OPTIONS entry in settings.py.

Example

DJANGO/djsuper/superheroes/templates/hero_escape.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero.name }}</title>
    <style>.blue { color: blue; } .red { color: red; }</style>
</head>
<body>
<h1>{{ hero.name }}</h1>

<h2>Secret Identity: {{ hero.secret_identity }}</h2>

<h3>Not safe: {{ html_fragment }}</h3>
<h3>Safe: {{ html_fragment | safe }}</h3>


</body>
</html>
```

The url tag

- Create links to views
- Not URL path dependent
- Uses (and requires) names in URL config

The url tag can be used to create a link to a view via a configured route. To do this, add a name parameter to the url in your URLconf. This name can then be used to create a URL that points to that view. The good thing is that if the name of the view changes, or the location within your app, the link will always be generated correctly.

Example

DJANGO/djsuper/superheroes/templates/hero_urls.html

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

{% block title %}{{ title }}{% endblock title %}

{% block content %}

<h1>Pick a hero (URL version):</h1>

{% for hero in superheroes %}
<h3>
    <a href="{% url 'superheroes:herodetails' hero.name  %}">{{ hero.name }}</a>
</h3>
{% endfor %}

{% endblock %}

</body>
</html>
```

Static files

- Unprocessed, unrendered files
- Images, CSS, JS
- Add static/appname to app folder
- Can also be served separately

Static files are those files which don’t require any processing, such as image, CSS, and JavaScript files.

Some sites serve those files separately (i.e., not with Django), using links that point to a particular URL on their site.

However, it is easy to serve static files with Django. Create a folder named static in the app folder. Put your static files there, and then use the get_static_prefix tag in URLs:

```
<img src="{% get_static_prefix %}app_name/image_file_name"/>
```

Example

DJANGO/djsuper/superheroes/templates/hero_static.html

```
{% extends "superheroes_base.html" %}
{% load static %}
{% block content %}

<h1>{{ hero.name }}</h1>
<h2>Secret Identity: {{ hero.secret_identity }}</h2>
<h2>Real Name: {{ hero.real_name }}</h2>
<h3>Powers: {{ hero.powers.all | join:", "}}</h3>
<h3>Enemies: {{ hero.enemies.all | join:", " }}</h3>
<br/>
<img
    src="{% get_static_prefix %}{{ image_name }}.jpg"
    alt="My image"
    width="400"
/>

{% endblock %}
```

Adding bootstrap

- Originally developed by Twitter
- Adds modern look
- Powerful layout tools
- Very configurable
- Use local or link to remote

bootstrap is a CSS library, originally developed at Twitter. It adds a modern look to web pages, but more importantly, provides simple but powerful layout tools.

Bootstrap is very configurable. We will only look at very basic Bootstrap features in this section. To get started, you need to include a Bootstrap link in your templates. If you are using a base template, you should put it there. The easiest thing to do is to specify a link that points to the bootstrap home page.

You can also download the bootstrap files and put them in your static folder. If you do this, and nothing else, it will add some nice styles to your pages.

See https://getbootstrap.com/getting-started/ for enhancing your bootstrap pages.

NOTE	When using the custom cookiecutter template, Bootstrap is already loaded in the base template named ***appname*.base.html** for your app.

Example

DJANGO/djsuper/superheroes/templates/superheroes_base.html

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <title>{% block title %}superheroes{% endblock title %}</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta name="description" content="">
    <meta name="author" content="">

    <!-- HTML5 shim, for IE6-8 support of HTML5 elements -->
    <!--[if lt IE 9]>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.3/html5shiv.min.js"></script>
    <![endif]-->

    {% block css %}
    <!-- Latest compiled and minified Bootstrap 4 beta CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/css/bootstrap.min.css" integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M" crossorigin=
"anonymous">

    <!-- Your stuff: Third-party CSS libraries go here -->

    {% endblock css %}

  </head>

  <body>

    <div class="container">

      {% block header %}{% endblock header %}
      {% block content %}
        <p>Default content for superheroes</p>
      {% endblock content %}
      {% block footer %}{% endblock footer %}

    </div> <!-- /container -->

    <!-- Javascript Placed at the end of the document so the pages load faster -->
```

```
    {% block javascript %}
      <!-- Required by Bootstrap v4 beta -->
      <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-
KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin=
"anonymous"></script>
      <script src
="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js" integrity
="sha384-b/U6ypiBEHpOf/4+1nzFpr53nxSS+GLCkfwBdFNTxtclqqenISfwAzpKaMNFNmj4" crossorigin
="anonymous"></script>
      <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/js/bootstrap.min.js" integrity="sha384-
h0AbiXch4ZDo7tp9hKZ4TsHbi047NrKGLO3SEJAg45jXxnGIfYzk4Si90RDIqNm1" crossorigin=
"anonymous"></script>

      <!-- Third-party javascript libraries go here -->

      <!-- App-specific Javascript in this file -->

      <script src="{% static 'js/superheroes.js'"></script>


    {% endblock javascript %}

  </body>
</html>
```

Chapter 7 Exercises

Exercise 7-1 (bands/*)

Create two templates – one to display a list of your bands, and one with the detail of each band. Use the base template provided, and have the two main templates include it. Modify the base template as desired.

Chapter 8: Querying Django Models

Objectives

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

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

DJANGO/djsuper/superheroes/viewsqueries.py

```
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, 'hero_queries.html', context)
```

Example

DJANGO/djsuper/superheroes/templates/hero_queries.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ hero_name }}</title>
</head>
<body>
<h1>{{ hero_name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>
</body>
</html>
```

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>]

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

Query Functions

Returning QuerySets	Returning Objects
filter()	get()
exclude()	create()
annotate()	get_or_create()
order_by()	update_or_create()
reverse()	latest()
distinct()	earliest()
values()	first()
values_list()	last()
dates()	
datetimes()	**Returning other values**
none() (empty)	bulk_create() (None)
all()	count() (int)
union()	in_bulk() (dict)
intersection() (1.11+ only)	iterator() (iterator)
difference() (1.11+ only)	aggregate() (dict)
select_related() (1.11+ only)	exists() (bool)
prefetch_related()	update() (int)
extra()	delete() (int)
defer()	as_manager() (Manager)
only()	
using()	
select_for_update()	
raw()	

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
```

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'}
```

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>]
```

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>]
```

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

Chapter 9: More about models

Objectives

- Define Meta classes for model configuration
- Create custom methods for objects
- Implement custom managers for models
- Control DB access by overriding standard methods

Customizing models

- Customize at object or model level
- Override DB functions

While the builtin functionality of Django covers most use cases, there are times when you need a model to do something that the Django developers didn't think of. When this occurs, you can customize models.

Use a nested Meta class to add configuration details to a model.

For custom row-level functionality, define custom methods on your models. These can be called from individual models.

If you need custom table-level functionality (filtering rows is a common need), define a new Manager and then define custom methods on the Manager. These methods will be available via model.objects, just like filter() and exclude().

You can also control what happens when an object is saved, updated, or deleted.

Example

DJANGO/djmore/superheroes/models.py

```
from django.db import models
import logging

logging.basicConfig(
    filename='superheroes.log',
    level=logging.INFO,
)

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

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

    def __str__(self):
        return self.name

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

    def __str__(self):
        return self.name

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

    def __str__(self):
        return self.name

class Superhero(models.Model):
    name = models.CharField(max_length=32)
    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)
    objects = SuperheroManager()

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

```
    class Meta():
        ordering = ['secret_identity']

    def get_brief_enemies(self):
        enemies = [e.name.split()[-1] for e in self.enemies.all()]
        return '/'.join(enemies)

    def save(self, *args, **kwargs):
        logging.info("Created superhero {}".format(self.name))
        super().save(*args, **kwargs)
        # do something else here as needed
```

Meta options

- Nest Meta class inside model
- Add config data

One easy way to customize models is to define a class named Meta inside the model, and assign values to class variables. This is Django's way of configuring the model.

NOTE The Meta class is completely unrelated to Python's concept of metaclasses, which Django uses, but not here.

Example

DJANGO/djmore/superheroes/views_meta.py

```
from django.shortcuts import get_object_or_404, get_list_or_404, render
from .models import Superhero

def hero_sort(request):
    heroes = get_list_or_404(Superhero)
    data = {
        'heroes': heroes
    }
    return render(request, 'hero_meta.html', data)

def hero_details(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero
    }
    return render(request, 'hero_details.html', data)
```

DJANGO/djmore/superheroes/templates/hero_meta.html

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

{% block content %}

<h1>Pick a hero:</h1>

{% for hero in Superhero %}
<h3><a
  href="{% url 'superheroes:herodetails' hero.name %}">
    {{ hero.name }}
</a>
    {{ hero.secret_identity }}
</h3>
{% endfor %}

{% endblock %}

</body>
</html>
```

Table 7. Meta options

Option	Default	Description
abstract	False	If True, model will be abstract base class
app_label	None	Declare which app model belongs to
base_manager_name	None	Name of manager to use
db_table	app_name + model_name	Table name to use for model
db_tablespace	DEFAULT_TABLESPACE setting	Tablespace to use
default_manager_name	objects	Name of default manager
default_related_name	modelname_set	Name used by related tables
get_latest_by	None	What field to use for latest() and earliest() methods
managed	True	If True, create/update database tables as needed
order_with_respect_to	None	Make object orderable based on field
ordering	None	List of fields to sort on (use field__name for related field)
permissions	None	Extra (custom) permissions (as 2-tuples)
default_permissions	('add', 'change', 'delete)	Default permissions for model
proxy	False	Subclass another model
required_db_features	None	Features required for particular DB connection
required_db_vendor	None	Model-specific DB vendor (one of sqlite, postgresql, mysql, oracle)
select_on_save	False	Use old (< 1.6) behavior on saves
unique_together	None	List of fields that must be unique when taken
index_together	None	List of fields that are indexed together
verbose_name	Munged version of model name	Human-readable name for one object
verbose_name_plural	Verbose name + 's'	Human-readable name for more than object (i.e., plural)

Custom methods

- Customize at object level
- Completely up to developer
- Available via objects

You can customize models at the object level by adding methods to a model, since a model is just a normal Python class. Methods have access to self, which represents one "row", as well as the model itself, which represents the table.

Example

DJANGO/djmore/superheroes/views_custom.py

```
from django.shortcuts import render, get_object_or_404
from .models import Superhero


def hero_custom(request, hero_name):
    hero = get_object_or_404(Superhero, name=hero_name)
    context = {
        'page_title': "Custom Function",
        'hero': hero,
        'enemies': hero.get_brief_enemies
    }
    return render(request, 'hero_custom.html', context)
```

DJANGO/djmore/superheroes/templates/hero_custom.html

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

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

<h2>{{ hero.name }}'s Enemies</h2>
{{ enemies }}

{% endblock %}
```

Custom Managers

- Customize at table level
- Inherit from standard manager
- Add table-wide methods

To provide a table-wide (aggregation) function, you can create a custom manager. It will inherit from the standard manager (**django.models.Manager**), and you can add any functions you need.

In a custom function, self is of course the manager itself, so you can call all(), filter(), exclude(), etc., from self.

Example

DJANGO/djmore/superheroes/views_manager.py

```
from django.shortcuts import render, get_object_or_404
from .models import Superhero


def hero_manager(request):
    fliers = Superhero.objects.get_fliers()
    context = { 'page_title': "Fliers", 'fliers': fliers }
    return render(request, 'hero_manager.html', context)

def hero_details(request, hero_name):
    hero =  get_object_or_404(Superhero, name=hero_name)
    data = {
        'hero': hero
    }
    return render(request, 'hero_details.html', data)
```

DJANGO/djmore/superheroes/templates/hero_manager.html

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

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

{% for hero in fliers %}
<h3>
    {{ hero.name }} ({{ hero.secret_identity }})
</h3>

{% endfor %}
{% endblock %}
```

Overriding standard methods

- Add functionality to standard DB methods
- Common functions to override: save(), delete()

Sometimes you will want to execute some logic when an object is saved or deleted. To do this, you can override the save() and delete() methods in the model.

Typically you would add the extra logic, then call the base version of the function, using super().

Chapter 9 Exercises

Exercise 9-1 (music/bands/models.py)

In your bands app:

- Change the default ordering to be by genre
- Create a custom object method get_genre() that returns the genre in uppercase; create a view and a template to use it.

Chapter 10: Forms

Objectives

- Create HTML forms
- Use different kinds of form widgets
- Process form data
- Validate form data

Forms overview

- User interface for apps
- Use HTML <form> tag
- Django goes beyond basics

In simplest terms, an HTML form is a collection of widgets inside a <form> tag. It is used to get information from the user. It is the primary user interface of web apps. (The other interface is the URL itself).

In addition to the standard HTML form input tags, Django provides more complex elements such as popup date pickers.

A form is associated with a URL that will process the data in the form. In Django, that URL maps to a view, as usual, and the view will typically display and process the form.

Forms are submitted – usually by the user pressing a "submit" button, or just pressing Enter/return. Forms can have more than one submit button, as well.

Handling forms can be tricky to get just right.

GET and POST

- HTTP methods
- GET sends data via URL
- POST sends data as content

When working with forms, they may be accessed via GET or POST methods. As a general rule, use GET for requests that do not change the state of the app, and POST for requests that modify the state of the app.

Don’t use GET for passwords, since GET data is visible on the URL.

For instance, use POST to submit registration data to a site, or use GET to query the site for information.

The Form class

- Describes form
- Contains fields, AKA widgets (input elements)
- Specifies action (URL for processing)
- A Form is a Django class that represents one HTML form. It contains fields that correspond to HTML input elements (AKA widgets).

Forms can be bound (filled-in with data) or unbound (empty). To check whether a form is bound or unbound, use the **method** attribute of the request object passed into a view. This will be either "GET" or "POST". If it is "GET", then the form is unbound, otherwise the form is bound, and contains data.

NOTE The ModelForm class maps a model to a form; this is used by the Admin app.

Building a form

- Inherit from django.forms.Form
- Create fields (similar to models)

To create a form, create a class that inherits from django.forms.Form. In the class, create class variables that are instances of various field types, which correspond to widgets on the form.

The field names are very similar to Model fields, although there are sometimes several form fields that all correspond to the same Model type, but with different validation.

Example

DJANGO/djform/superheroes/forms.py

```
#!/usr/bin/env python
# (c) 2016 CJ Associates
#
from django import forms
from .models import Superhero
from .validators import small_integer_only

class LittleIntegerField(forms.IntegerField):
    default_validators = [small_integer_only]


class DemoForm(forms.Form):
    demo_boolean = forms.BooleanField()
    demo_char = forms.CharField(max_length=10, strip=True)
    demo_choice = forms.ChoiceField(choices=[(1, 'A'), (2, 'B'), (3, 'C')])
    demo_date = forms.DateField(label="Date")
    demo_email = forms.EmailField(label="Electronic mail address:")
    demo_float = forms.FloatField(help_text="Please enter a floating point number")
    demo_int1 = LittleIntegerField()
    demo_int2 = LittleIntegerField()
    demo_regex = forms.RegexField(regex=r'(?i)^a[a-z]{1,5}$')
    # submit = forms

    # add clean function here...
    def clean_demo_boolean(self):
        bool = self.cleaned_data['demo_boolean']
#        raise forms.ValidationError("That is an invalid Boolean")
        return  not bool


COLORS = 'green red blue purple orange'.split()
COLOR_CHOICES = [(c.title(), c) for c in COLORS]


class HeroForm(forms.Form):

    hero_name = forms.CharField(label='Hero', max_length=40)
    hero_color = forms.ChoiceField(
        label="Color",
        choices=COLOR_CHOICES,
    )
```

```
class HeroModel(forms.ModelForm):
    class Meta():
        model = Superhero
        fields = ['name', 'real_name', 'city', 'secret_identity']
        labels = {
            'name': 'Hero Name',
            'city': 'City where they hang out',
        }
```

Field types

- Data entry elements
- Wide variety
- Fields take custom parameters

Django provides a wide variety of form field elements. Most of these map to the equivalent Model field type. Many field types have field-specific parameters, and there are some common parameters accepted by all field types.

Table 8. Common Field Parameters

Parameter	Description
required	Whether field is required, defaults to True
label	What will be displayed on the form
label_suffix	Specify suffix for label (default suffix is " =")
initial	Initial value for field
help_text	Help text for the field (displayed next to field)
error_messages	Dictionary of custom error message

NOTE Django also supports custom fields.

Table 9. Form Fields

Field	Python Type	Empty Value
BooleanField	boolean	False
CharField	str	'' (empty string)
ChoiceField	str	'' (empty string)
TypedChoiceField	str	Specified by empty_value parameter
DateField	datetime.date	None
DateTimeField	datetime.datetime	None
DecimalField	decimal	None
DurationField	datetime.timedelta	None
EmailField	str	'' (empty string)
FileField	UploadedFile	None
FilePathField	str	None
FloatField	float	None
ImageField	UploadedFile	None
IntegerField	int	None
GenericIPAddressField	str	'' (empty string)
MultipleChoiceField	list	[]
TypedMultipleChoiceField	list	Specified by empty_value parameter
NullBooleanField	bool	None
RegexField	str	'' (empty string)
SlugField	str	'' (empty string)
TimeField	datetime.time	None
URLField	str	'' (empty string)
UUIDField	UUID	'' (empty string)
ComboField	str	'' (empty string)
MultiValueField	varies	'' (empty string)
SplitDateTimeField	datetime.datetime	None
ModelChoiceField	A model instance	None
ModelMultipleChoiceField	QuerySet	Empty QuerySet

Templates and forms

- Template provides <form> tag
- Provide form object as template variable inside <form>
- Add <submit> field

To use the form, create a template, and in the template create a <form> tag. Specify method="post" and action="/path/to/form/view". Be sure to add a SUBMIT button.

Pass the form instance in as part of the template context, and just specify it as a template variable. The template will render the fields of the form.

By default, the fields are rendered without formatting, so you will typically want to call one of the customer renderers, such as form.as_p.

Example

DJANGO/djform/superheroes/templates/form_demo.html

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

{% block content %}
<h1>{{ page_title }}</h1>
{% if invalid %}
<div class="btn-warning">
    Some fields need to be corrected
</div>
{% endif %}
<form action="/superheroes/demoform" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Submit"/>
</form>

{% endblock %}
```

DJANGO/djform/superheroes/templates/form_results.html

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

{% block content %}
<h1>{{ page_title }}</h1>
<div class="container">
{% for field, value in data.items %}
    <div class="row">
        <div class="col-2">{{ field }}</div>
        <div class="col-10">{{ value }}</div>
    </div>
{% endfor %}
</div>
{% endblock %}
```

DJANGO/djform/superheroes/templates/hero_details.html

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

{% block content %}

<h1 style="color:{{ color }};">{{ hero.name }}</h1>
<h2>Secret Identity: {{ hero.secret_identity }}</h2>
<h2>Real Name: {{ hero.real_name }}</h2>
<h3>Powers: {{ hero.powers.all | join:", "}}</h3>
<h3>Enemies: {{ hero.enemies.all | join:", " }}</h3>

{% endblock %}
```

DJANGO/djform/superheroes/templates/hero_select.html

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

{% block content %}
<h1>{{ page_title }}</h1>
<form action="/superheroes/heroform/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Search"/>

</form>

{% endblock %}
```

DJANGO/djform/superheroes/templates/hero_select_p.html

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

{% block content %}
<h1>{{ page_title }}</h1>
<form action="/forms/select/" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Search"/>

</form>

{% endblock %}
```

```
*{examplefile}*
```

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

{% block content %}
<h1>{{ page_title }}</h1>
<form action="/forms/select/" method="post">
    {% csrf_token %}
    <table border="2">
    {{ form.as_table }}
    </table>
    <br/>
    <input type="submit" value="Search"/>
</form>

{% endblock %}
```

```
*{examplefile}*
```

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

{% block content %}
<h1>{{ page_title }}</h1>
<form action="/forms/select/" method="post">
    {% csrf_token %}
    <ul>
    {{ form.as_ul }}
    </ul>
</form>

{% endblock %}
```

Processing the form

- Display form
- Get data from fields
- Redraw form or render result page

The form will be processed when the user submits the form. It will be sent as POST data to whatever method (usually in views.py) you have specified.

In the view, you can test whether the form is being displayed for the first time (unbound) or being submitted (bound) by checking whether the request method is GET or POST. If the request method is GET, then it will display an empty form. If POST, then the renderer will create the formed filled in with the submitted data.

If the data is invalid, is_valid() method will return True, and the , the form will be redrawn. If is_valid() returns True, then you can process the data in the form as needed.

Example

DJANGO/djform/superheroes/views.py

```
"""
    Views for the DJForms Project

    These are forms illustrating how forms work in Django
"""
from django.shortcuts import get_object_or_404, render
from .forms import DemoForm, HeroForm, HeroModel
from .models import Superhero

def home(request):
    """
    Welcome page

    :param request: HTTP request
    :return: HTTP Response
    """
    data = {
        'message': 'Welcome to the superheroes app for forms',
    }
    return render(request, 'home.html', data)


def demoform(request):
    """
    Generic form demo with various fields

    :param request: HTTP request
    :return: HTTP Response
    """
    invalid = False

    if request.method == 'POST':
        form = DemoForm(request.POST)
        if form.is_valid():
            # if data is valid, show results page
            context = {
                    'page_title': 'Form Fields Results',
                    'data': form.cleaned_data,
            }
            return render(request, 'form_results.html', context)
        else:
            # show form with errors for correcting
            invalid = True
```

```
    else:
        form = DemoForm() # unbound form

    # unless POST/valid, redraw form
    context = {
        'page_title': 'Form Fields Example',
        'form': form,
        'invalid': invalid,
    }
    return render(request, 'form_demo.html', context)


def heroform(request):
    """

    :param request: HTTP request
    :return: HTTP Response
    """

    # bound (filled-in) form
    if request.method == 'POST':
        form = HeroForm(request.POST)
        if form.is_valid():
            hero_name = form.cleaned_data['hero_name']
            hero_color = form.cleaned_data['hero_color']
            request.session['color'] = hero_color
            hero = get_object_or_404(Superhero, name=hero_name)
            context = {
                'page_title': 'Hero Details',
                'hero': hero,
                'color': hero_color,
            }
            return render(request, 'hero_details.html', context)

    else:
        # unbound (empty) form
        form = HeroForm()

        context = {
            'page_title': 'Form Example',
            'form': form,
        }
        return render(request, 'hero_select.html', context)


def heromodel(request):
    """
```

```
    :param request: HTTP request
    :return: HTTP Response
    """

    # bound (filled-in) form
    if request.method == 'POST':
        form = HeroModel(request.POST)
        if form.is_valid():
            context = {
                'page_title': 'Hero Details',
                'name': form.cleaned_data['name'],
                'secret_identity': form.cleaned_data['secret_identity'],
                'real_name': form.cleaned_data['real_name'],
            }
            # form.save()
            return render(request, 'hero_model_results.html', context)

    else:
        # unbound (empty) form
        form = HeroModel()

        context = {
            'page_title': 'Form Example',
            'form': form,
        }
        return render(request, 'hero_model_select.html', context)
```

Validation

- Prevent invalid form entry
- Can be client- or server-side

The various field types will automatically validate for the type. To accept a particular string, you can use the RegexField, which only accepts a string matching a specified regular exception.

You can add a list of custom validation functions for any form field with the **validators** parameter. will

Example

```
from django.db import models

class SomeModel(models.Model):
    my_field = models.IntegerField(validators=[validate_whatever])
```

Using ModelForm

- Helper class for creating forms from models
- No form coding required
- Use Meta class to add configuration

Since many apps are based on databases, the **forms** module provides a **ModelForm** class which makes it easy to create forms with the same fields as models.

To use a ModelForm, create a class that inherits from ModelForm. Within that class, define a class named Meta which will contain configuration information. The form needs at least the model name and a list of fields to add to the form. You may not want all fields in the model to be added to the form.

To use the form, create an instance of the derived class like any other form. To create an update-type form, pass an object from the model to the from constructor's **instance** parameter.

Example

DJANGO/djform/superheroes/templates/hero_model_select.html

```
{% extends "superheroes_base.html" %}
{% load crispy_forms_tags %}

{% block content %}
<h1>{{ page_title }}</h1>
<form action="{% url 'superheroes:heromodel' %}" method="post">
    {% csrf_token %}
    {{ form | crispy }}
    <input type="submit" value="Search"/>

</form>

{% endblock %}
```

DJANGO/djform/superheroes/templates/hero_model_results.html

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

{% block content %}

<h1>{{ name }}</h1>
<h2>Secret Identity: {{ secret_identity }}</h2>
<h2>Real Name: {{ real_name }}</h2>

{% endblock %}
```

Beyond the basics

- Change form layout
  - form.as_table
  - form_as_p
  - form_as_ul

Out of the box, the forms rendered by Django are not very attractive. Of course you can use CSS to improve things, but the Form class provides several methods to render them in a nicer way.

The first is .as_table, which provides the labels and entry fields as table rows. You need to wrap the <table> tags around the form.

The second is .as_p, which puts <p> (paragraph) tags around each label/entry.

Finally, .as_ul, wraps <li> tags around each label, so you can put the form inside either <ol> or <ul> tags.

Crispy Forms

- Make more attractive forms
- Control layout from the view
- Reusable layouts

Even using the layouts on the previous page does not create the most attractive forms. For making attractive and consistent forms, you can use the crispy-forms package. This moves control of HTML generation into the view, and allows for reusable layouts. This can be combined with class-based views to inherit common form features.

See the details at https://django-crispy-forms.readthedocs.io/en/latest/index.html

Chapter 10 Exercises

Exercise 10-1 (music/bands/*)

Create a search form with entries for band name and genre. On submit, search your database and return the details for any matching bands.

Chapter 11: Debugging and trouble-shooting

Objectives

- Read the Django debug page
- Learn common Django troubleshooting techniques
- Implement and use the Django Debug Toolbar

Debugging

- Print statements
- Django debug page
- Django debug toolbar

Django Debug page

- For web apps (not services)
- Lots of info

For web apps, when an exception occurs, you get the Django debug page. It contains tons of useful information.

The information in the debug page includes:

- Exception details
- Exception location
- Traceback
  - Local vars
- Request information

Django Debug Toolbar

- Appears as narrow strip on right of page
- Click to expand

The Django Debug Toolbar gives much more information than the default debug page. It is available in all rendered web pages, whether or not there is an exception. It appears as a narrow clickable graphic on the right side of the screen. Once you click on it, it expands into a menu with many useful choices for information about the page.

Setting breakpoints in your code

- Import pdb
- pdb.set_trace()

If you are using the Python debugger, it can be tricky to step through all the code to get to a particular point. You can import pdb, and then call pdb.set_trace to programmatically set a breakpoint at the desired location.

Example

```
import pdb
...
pdb.set_trace()
...
```

Chapter 11 Exercises

Exercise 11-1 (T/B/D)

Chapter 12: Class-based views

Objectives

- Understand class-based views
- Discover builtin generic views
- Implement class-based views

Class-based views

- Defined as classes
- Actual views are methods
- Allow for more flexibility
- Good for requests with multiple HTTP request types

Class-based views are views implemented as classes. The actual view functions are implemented as methods.

Django has some builtin generic class-based views for working with Django models.

Such views can be very flexible, using inheritance to isolate shared behavior. You can also define separate methods for GET, PUT, POST, and other HTTP requests. You can inherit from generic builtin views and only change what you need.

NOTE

Audrey and Daniel Greenfeld, authors of *2 Scoops of Django*, have the following philosophy: "Fat models, thin views, and stupid templates".

This means that most of the business logic should stay at the source, i.e. the models. Views should do the minimum necessary to hook data up with templates, and templates should have only display logic, never business logic.

Generic views

- Predefined view classes
- No reinventing wheels
- Cover frequent situations
- Provide structure and encapsulation
- Designed to be subclassed

When creating a Django app, you'll see that a lot of views are very similar. You get data from the request, build a context, and render a template. You can end up with a lot of duplicate code.

To avoid some of this duplication, Django provides generic views. These are predefined class-based views that work with Django datasets and templates to take care of the "busywork" of presenting data. They are appropriate for a variety of application types. They factor out the "boilerplate" code that is part of all views.

All generic views ultimately inherit from the **View** class.

Table 10. Types of generic views

View	Description
Base views	
View	Generic generic view
TemplateView	Render URL parameters
RedirectView	Redirect to specified URL
Generic display views	
DetailView	Display data from object
ListView	Display data from list (typically QuerySet)
Generic editing views	
FormView	Display a form
CreateView	Display a form for creating an object
UpdateView	Display a form for updating an object
DeleteView	Display a form for deleting an object
Generic date views	
ArchiveIndexView	Display objects by date
YearArchiveView	Display objects for one year
MonthArchiveView	Display objects for one month
WeekArchiveView	Display objects for one week
DayArchiveView	Display objects for one day
TodayArchiveView	Display objects for today
DateDetailView	Display one object by date

Example

DJANGO/djclass/superheroes/urls.py

```
"""
URL Configuration for superheroes
"""
from django.conf.urls import url
from django.urls import path
from django.views.generic import TemplateView
from . import views

urlpatterns = [
    # welcome page, no class-based views
    path(
        '',
        views.HomeView.as_view,
        name = 'home'
    ),

    # NO view -- don't do this:
    path(
        'noview',
        TemplateView.as_view(template_name='noview.html'),
        name="noview",
    ),

    # minimal views with models
    path(
        'minimallist',
        views.HeroListViewMinimal.as_view(),
        name="minimallist",
    ),
    url(
        r'(?i)minimaldetails/(?P<pk>\d+)$',
        views.HeroDetailViewMinimal.as_view(),
        name="minimaldetails",
    ),

    #
    url(
        r'(?i)genericcontext$',
        views.GenericContext.as_view(),
        name="genericcontext",
    ),
    url(
        r'(?i)genericlist$',
```

```
        views.HeroListView.as_view(),
        name="genericlist",
    ),
    url(
        r'(?i)genericdetail/(?P<pk>\d+)/$',
        views.HeroDetailView.as_view(),
        name="genericdetail",
    ),
    url(
        r'(?i)herocreate/$',
        views.HeroCreateView.as_view(),
        name="herocreate",
    ),
    url(
        r'(?i)heroupdate/(?P<pk>\d+)/$',
        views.HeroUpdateView.as_view(),
        name="heroupdate",
    ),
    path(
        'success', views.SuccessView.as_view(), name="success",
    )
]
```

DJANGO/djclass/superheroes/views.py

```
from django.views.generic import (
    TemplateView, ListView, CreateView, DetailView, UpdateView
)
from django.urls import reverse_lazy
from django.shortcuts import render
from .models import Superhero, City


class HomeView(TemplateView):
    template_name = 'home.html'
    data = {
        'message': 'Welcome to the superheroes app for class-based views',
    }

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context.update(data)
        return context

class HeroListViewMinimal(ListView):
    model = Superhero

class HeroDetailViewMinimal(DetailView):
    model = Superhero

class HeroListView(ListView):
    context_object_name = 'heroes'
    model = Superhero

class HeroDetailView(DetailView):
    context_object_name = 'hero'
    model = Superhero

class HeroCreateView(CreateView):
    model = Superhero
    fields = ['name', 'real_name', 'secret_identity', 'city']
    success_url = reverse_lazy('superheroes:success')

class CityCreateView(CreateView):
    model = City
    fields = ['name']
    success_url = reverse_lazy('superheroes:success')

class HeroUpdateView(UpdateView):
    model = Superhero
```

```
    # template = "hero_update.html"
    fields = ['name', 'real_name', 'secret_identity', 'city']
    success_url = reverse_lazy('superheroes:success')

class SuccessView(TemplateView):
    template_name = 'success.html'
```

Using plain generic views

- Create in URLconf
- call as_view()

To use one of the generic views as-is, just create it in the URL configuration. You don't need to write a separate view. Because the URLConf expects a function object, you can't pass in the view object directly. You must call as_view() to return a function that acts like a view.

CAUTION	Using a generic view in this way is not recommended, because it moves display logic into the URL configuration. Such a decision should remain in the view.

Example

DJANGO/djclass/superheroes/templates/superheroes/generic_only.html

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

{% block content %}

<h1>Welcome to a generic page</h1>

<h3>Rendered by a generic class without any code in views.py</h3>

{% endblock %}
<q1>`   </q1>
```

Subclassing Generic Views

- Create new view
- Add class variables as needed

When using generic views, it it most common to subclass one of the builtin views. Then you can add attributes and functions as needed, such as the template name. This keeps the URLConf cleaner and neater, and lets you create inheritable views which share similar features.

Class attributes may be defined in the class, or passed as attributes to the as_view() method.

You can overwrite methods named get, post, head, etc. in the view to response to those HTTP requests specifically.

Example

DJANGO/djclass/superheroes/templates/generic_view.html

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

{% block content %}

<h1>Welcome to a generic page</h1>

<h3>Rendered by a generic view class with minimal code in views.py</h3>

{% endblock %}
<q1>`   </q1>
```

Passing variables to a template

- Define variables in view class
- Override get_context_data()

To pass variables into a template when using generic classes, first define the variables in the class. Then override get_context_data(). First call the base class get_context_data() to get the default context, then update that with your class variables. Finally, return the updated context.

Example

DJANGO/djclass/superheroes/templates/generic_context.html

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

{% block content %}

<h1>Welcome to a generic page</h1>

<h3>Rendered by a generic view {{ message }}</h3>
<br/>
<ul>
    {% for fruit in fruits %}
    <li>{{ fruit }}</li>
    {% endfor %}
</ul>


{% endblock %}
<q1>`   </q1>
```

ListView and DetailView

- Work directly with QuerySets
- DetailView displays one object
- ListView displays list of objects

Because many apps just display data from the app's database, there are two generic views for passing an object or a QuerySet to a template. These can be subclassed, so you can easily add custom attributes and context variables.

To inherit from ListView, just specify the template name and the model name. It is usually convenient to define the name the QuerySet will use in the template, via context_object_name.

To use the DetailView, specify the template and model name, and define the pk parameter in the URL. DetailView is hard-wired to use the pk field.

Example

DJANGO/djclass/superheroes/templates/hero_list.html

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

{% block content %}

<h1>Pick a hero:</h1>

{% for hero in heroes %}
<h3><a
  href="{% url 'superheroes:genericdetail' hero.id %}">
    {{ hero.name }}
</a>
    {{ hero.secret_identity }}
</h3>
{% endfor %}

{% endblock %}

</body>
</html>
```

DJANGO/djclass/superheroes/templates/hero_details.html

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

{% block content %}

<h1>{{ hero.name }}</h1>
<h2>Secret Identity: {{ hero.secret_identity }}</h2>
<h2>Real Name: {{ hero.real_name }}</h2>
<h3>Powers: {{ hero.powers.all | join:", "}}</h3>
<h3>Enemies: {{ hero.enemies.all | join:", " }}</h3>

<a href="{% url 'superheroes:genericlist' %}">Return to list</a>

{% endblock %}
```

Chapter 12 Exercises

Exercise 12-1 (music/bands/*)

In your bands project, create two generic views – one to list bands, and the other to display the details about one band specified by its primary key.

Chapter 13: Django User Authentication

Objectives

- Understand Django's builtin authentication system
- Create and use User objects
- Add authentication to web requests
- Customize users

Users

- Core of auth system
- Represent app users
- All users created equal

User objects represent the users of your application. Via Users, you can set access, keep profiles, tag content with its creator, and other tasks. There is only one type of User; elevated permissions ("root", "admin", etc.) are controlled by User attributes.

Users have the following attributes:

- username
- password
- email
- first_name
- last_name

Creating Users

- Use the admin tool
- User.objects.create_user()
- manage.py createsuperuser ...
- manage.py changepassword ...

If you have enabled the Django admin system, then you can easily create users through that interface.

You can also create users with the create_user() helper function.

As with most auth systems, the password itself is not stored in the database, but just the hash value of the password. One result of this is that you should not directly modify the password field in the database.

You can create a superuser (admin user) with

```
manage.py createsuperuser --username=username email=email
```

You can change a password with

```
manage.py changepassword username
```

Alternatively, you can change the password with the set_password() method on a User object.

Authenticating Users (low-level)

- django.contrib.auth.authenticate
- Pass in username/password
- Returns non-None value on success

You can authenticate credentials with the authenticate() function from django.contrib.auth. It takes a username and password as parameters, and returns None if the authentication fails. It returns a non-None (i.e., True) value when the authentication is successful.

However, most of the time you will want to use the login_required() decorator, which calls the above for you.

Example

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # A backend authenticated the credentials
else:
    # No backend authenticated the credentials
```

Permissions

- Kinds of permission: add, change, delete
- Set permissions on object types and instances
- user.user_permissions.set(), .add(), .remove(), .clear()

You can set permissions on objects in general, and on specific instances of object.

You can also create custom permissions

Groups

- Users sharing permissions
- User gets all permissions of group

You can create any number of groups that represent a shared set of permissions.

Create groups via the Django admin page, or programmatically with django.contrib.auth.models.Group. Use

```
Group.objects.get_or_crate(name='group name')
```

To create a new group. It returns a tuple containing the new group object (if created) and a Boolean value that says whether the group was successfully created. After the group has been created, you can refer to it by name when working with users and permissions.

Web request authentication

- Django has builtin sessions
- request object has user attribute
- Check request.user.is_authenticated

Django uses sessions and builtin middleware to provide authentication for request objects.

A request.user attribute on every request corresponds to the current user. If the current user has not logged in, this attribute will be set to an instance of AnonymousUser, otherwise it will be an instance of User.

To tell whether the user is authenticated, use:

```
if request.user.is_authenticated:
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

login shortcuts

- login_required() decorator
- LoginRequiredMixin
- Decorator for view functions

To simplify working with authentication, there are two shortcuts – one for view functions, and one for class-based views.

For view functions, use the @login_required decorator. When a user visits a url served by the decorated view function, it redirects to settings.LOGIN_URL (a path to your app's login page) if the user is not logged in.

For class-based views, you can add LoginRequiredMixin to the list of base classes.

from django.contrib.auth.mixins import LoginRequiredMixin

Example

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

or

```
from django.contrib.auth.mixins import LoginRequiredMixin

class MyView(LoginRequiredMixin, View):
    login_url = '/login/'
    redirect_field_name = 'redirect_to'
```

Customizing users

- Proxy user
- One-to-one
- Extend builtin user

There are three primary ways to customize a Django user.

Proxy model

A proxy model can be used when you need to modify the behavior of the user model without actually changing the database schema. You could, for instance, specify a different manager for queries, or add custom methods.

To do this,

1. import the User model from `django.contrib.auth.models`.
2. define a new model that inherits from User
3. modify the model
    - add a different manager
    - define custom methods
    - add Meta fields

Think of a proxy model as a *view* of the "real" User model.

One-to-one model

A one-to-one model can be used when you want to store extra data for users, but don't need to modify the authorization process. without the complexity of extending the user model.

This is most useful when there is profile data to be kept for each user.

To do this,

1. define a new model (do not inherit from `django.contrib.auth.models.User`)
2. add a field of type OneToOneField.
3. use *signals* to synchronize the new model with User
    - import `post_save` or other signals from `django.db.models.signals`
    - import `receiver` from `django.dispatch`
    - define methods for saving, updating, etc.

- decorate methods with `@receiver(post_save, sender=User)`
- methods will be called when User is updated

NOTE	Using a one-to-one model may result in extra queries to the database, which could have an adverse effect on throughput.

Custom model

The most flexible (and complex) way to customize a user is to extend the user model. In this case, you're creating a new model that inherits from User.

This can be used to completely customize the behavior of a user. If you do this, you must update the settings files with:

```
AUTH_USER_MODEL = '<myapp>.User'
```

Chapter 13 Exercises

Exercise 13-1 (dogs/*)

Add a simple login page to your dogs app. Use a form with Username/Password fields and a Submit button. You can use Django's builtin auth system.

Chapter 14: Session management

Objectives

- Understand sessions
- Enable sessions management
- Get/Retrieve user data

L

About sessions

- Store/retrieve data on site visitors
- Stored on server side
- Manages cookies

For security, Django implements sessions to manage cookies. You can store and retrieve user-specific data.

User data itself is not stored in cookies; the cookie contains a session ID. By default, user data is stored in your database, but there are other session backends you can use.

Enabling sessions

- Should already be enabled
- Configure in settings.py
  - MIDDLEWARE
  - INSTALLED_APPS
- request has session attribute

Sessions should already be enabled in your project's settings.py file.

The MIDDLEWARE setting needs to contain 'django.contrib.sessions.middleware.SessionMiddleware'. This should already be in place.

If for some reason it is not, add 'django.contrib.sessions.middleware.SessionMiddleware' to MIDDLEWARE and add 'django.contrib.sessions' to INSTALLED_APPS.

Types of session backends

- database
- file
- cache (e.g. Redis)

By default, session information is saved in your project database, but it can also be stored in a file or cache (e.g. Redis). You can configure a different backend by using the SESSION_ENGINE setting.

Be sure to run

```
manage.py migrate
```

to make sure the session database is set up.

CAUTION	Be sure 'django.contrib.sessions' is added to INSTALLED_APPS.

Accessing sessions from views

- Use request.session
- Modify anywhere in view

To access session information in a view, you can just access the request.session object, which is similar to a normal Python dictionary. You can make changes to the session, and they will be propagated to other views. It is sort of a global variable for the user, implemented as a dictionary-like object. Normal dictionary operations such as [] lookup work fine on a session object.

Table 11. request.session attributes

Attribute	Description
Standard dictionary methods	
get(key, default=None)	
pop(key, default=__not_given)	
keys()	
items()	
setdefault()	
clear()	
Session-specific methods	
flush()	Delete current session data
set_test_cookie()	Sets a test cookie :-)
test_cookie_worked()	Returns True if cookie accepted, else False
delete_test_cookie()	Delete test cookie
set_expiry(value)	Sets the expiration time for the session
get_expiry_age()	Return number of seconds until session expires
modification	last session modification time
expiry	expiry information for the session
get_expiry_date()	Return date session will expire
get_expire_at_browser_close()	Return True or False whether cookie expires when browser is closed
clear_expired()	Remove expired sessions
cycle_key()	Create a new session key with current session data

Chapter 15: Migrating Django Data

Objectives

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

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.

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

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.

NOTE	migrations are applied project-wide by default.

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

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

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 0001`.

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

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 15 Exercises

Exercise 15-1 (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).

Chapter 16: Configuration

Objectives

- Explore how Django settings work
- See a typical Django configuration
- Understand security considerations
- Implement 12 Factor settings

About Django configuration

- Every project needs configuration
  - Installed applications
  - Middleware
  - Database access
  - Deployment type
    - Development
    - Testing
    - Staging
    - Production
  - File Locations
    - Templates
    - Static files

A Django project needs a fair amount of **configuration** to let the variable Django modules know what to do. Ultimately these will be loaded into a global object named **django.settings**.

Settings and security

- Different settings for different deployments
- Keep passwords out of settings
- Store secure data outside project
- Use Django-environ

For convenience, `manage.py startproject` puts all of your project's configuration settings in one python module — **settings.py**. This is a bad idea.

The settings for development will be different from production settings. Developers will not be using the production database, for one thing. There needs to be an easy way to switch config for different kinds of deployments.

Some settings are sensitive (e.g. passwords, API keys), and you don't want them to be put under version control.

The following things might be different among development, staging, and production:

- database connection: database, user name and password
- API keys
- private encryption keys
- debug settings

In general, the solution is to keep the sensitive or deployment-specific settings in a separate place. It could be a config file that is read at the top of settings.py. You can create one version of the file for development, and another for production.

There are several approaches for this. You can make up your own scheme, or use a package such as **django-environ**.

Locating the settings module

- No "standard" location
- Set via DJANGO_SETTINGS_MODULE
- Set by **manage.py**

Surprisingly, there is no "standard" location for Django settings, or even an official module name. As noted, `django-admin startproject` creates a file named **settings.py**, but it can be named anything.

The settings module is named by the environment variable **DJANGO_SETTINGS_MODULE**.

The **cookiecutter-django** template creates a base file, then adds local or production settings to it.

Tools to make configuration easier

- django-admin
- django-environ
- cookiecutter

There are several tools to make Django configuration easier. The default tool, **django-admin**, comes with the basic Django installation. Using its **startproject** command creates a default setup, but is not really suitable for real-life projects.

django-environ is a tool that implements the concepts from **12 Factors**. It is described in the next section. It uses environment variables (or a .env file) to configure the Django settings.

You have been using cookiecutter throughout the course to set up a project. The cookiecutter template provided, cookiecutter-DJANGO, is a simplified version of the "standard" cookiecutter template, called **cookiecutter-django**, and available from XXX. This template provides many setup features for a full project.

If you need to create many projects, you will want to create your own cookiecutter template.

About django-environ

- Based on 12 Factors development
- Read from environment or file
- Keep credentials out of source code control
- Very flexible

The Django-environ extension provides an organized way to provide configuration values from either the environment, a separate file, or both.

Django-environ is an extension for managing configuration information. It reads configuration from environment variables or a configuration file. If you read from a file, it can be in the root folder of your project, or in a totally separate folder.

If the config file is within your project, be sure to exclude it from source code control.

It is a simple package that has some predefined labels for common Django settings.

To use, you import it in settings.py and create an Env object, then use that object to provide configuration, rather than hard-coding it in the settings file.

You can specify URIs for database, cache, mail, and search credentials. Django-environ will automatically parse the URI and create the needed fields.

For example:

```
DATABASE_URL=postgresql://postgres:scripts@localhost
```

becomes

```
{
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'PORT': None,
    'HOST': 'localhost',
    'NAME': '',
    'PASSWORD': 'scripts',
    'USER': 'postgres'
}
```

Example DB URIs

```
sqlite:////full/path/to/your/file.sqlite

postgres://username:password@hostname:port/database

mysql://username:password@hostname:port/database

oracle://username:password@hostname:port/database
oracle://username:password/DSN
oracle:// username:password/TSN
```

NOTE Django-environ is based on the 12 Factors of Web or SaaS development, created by the Heroku developers. See more at https://12factor.net/.

Implementing Django-environ

- Create config file (optional)
- Import in settings.py
- Create Env() instance
- Read environment via Env.read_env()
- Use instance to populate config variables

To use Django-environ, first import it in settings.py. The name to import is environ.

If you are reading config from a file, you can use Env to calculate the path.

Create an instance of environ.Env, then call read_env() from the instance. If you are using a file, pass the file name into read_env(). (It will default to .env in the same folder as settings.py).

You can move the secret key generated by django-admin startproject into the .env file, or add it as an environment variable.

Example

hello/hello/.env

```
DEBUG=on
DATABASE_URL=postgresql://postgres:scripts@localhost
SECRET_KEY=36f%sx6r(^&5=%7n*2n9*9v4c^t!p=*m9mgq1*5^b@47r=v07_
```

hello/hello/settings.py

```
...
import environ
...
env = environ.Env()
env.read_env()
...
SECRET_KEY = env('SECRET_KEY')
...
DATABASES = {
    'default': env.db(),
}
```

Django-environ supported types

- str
- bool
- int, float
- json
- list (FOO=a,b,c)
- tuple (FOO=(a,b,c))
- dict (BAR=key=val,foo=bar) #environ.Env(BAR=(dict, {}))
- url
- path (environ.Path)
- db_url
  - PostgreSQL: postgres://, pgsql://, psql:// or postgresql://
  - PostGIS: postgis://
  - MySQL: mysql:// or mysql2://
  - MySQL for GeoDjango: mysqlgis://
  - SQLITE: sqlite://
  - SQLITE with SPATIALITE for GeoDjango: spatialite://
  - Oracle: oracle://
  - LDAP: ldap://
- cache_url
  - Database: dbcache://
  - Dummy: dummycache://
  - File: filecache://
  - Memory: locmemcache://
  - Memcached: memcache://
  - Python memory: pymemcache://
  - Redis: rediscache://

- search_url
  - ElasticSearch: elasticsearch://
  - Solr: solr://
  - Whoosh: whoosh://
  - Xapian: xapian://
  - Simple cache: simple://
- email_url
  - SMTP: smtp://
  - SMTP+SSL: smtp+ssl://
  - SMTP+TLS: smtp+tls://
  - Console mail: consolemail://
  - File mail: filemail://
  - LocMem mail: memorymail://
  - Dummy mail: dummymail://

Example

hello/hello/.env

```
DEBUG=on
DATABASE_URL=postgresql://postgres:scripts@localhost
SECRET_KEY=36f%sx6r(^&5=%7n*2n9*9v4c^t!p=*m9mgq1*5^b@47r=v07_
```

hello/hello/settings.py

```
...
import environ
...
env = environ.Env()
env.read_env()
...
SECRET_KEY = env('SECRET_KEY')
...
DEBUG = env('DEBUG')
...
DATABASES = {
    'default': env.db()
}
```

hello/hello/views.py

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('<h2>Hello Django World</h2>')
```

hello/hello/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from hello.views import hello

urlpatterns = [
    url(r'^$', hello),
    url(r'^admin/', admin.site.urls),
]
```

Using cookiecutter config

- Use **cookiecutter-django**
- Includes settings for django-environ
- Private settings
  - .env
  - Environment variables

When using the **cookiecutter-django** template, you can either put settings in the environment, or create a file named .env. The template comes with a sample file named **example.env** to get you started.

Chapter 16 Exercises

Exercise 16-1 (DJANGO/tollhouse)

Create a new Django project named tollhouse. Instead of the using cookiecutter-DJANGO as we have been doing, use cookiecutter-django. This is the default cookiecutter file provided by the cookiecutter developers. Just hit enter on most of the prompts.

In *project*/config/base.py, change

```
# .env file, should load only in development environment
READ_DOT_ENV_FILE = env.bool('DJANGO_READ_DOT_ENV_FILE', default=False)
```

to

```
# .env file, should load only in development environment
READ_DOT_ENV_FILE = env.bool('DJANGO_READ_DOT_ENV_FILE', default=True)
```

NOTE	You could also set the environment variable DJANGO_READ_DOT_ENV_FILE instead of changing base.py.

Copy env.example to .env and change values as needed.

You will have to set up the configuration for your database to use SQLite.

Use either cookiecutter or `manage.py startapp` to create a simple app with just a "hello world" page.

Chapter 17: About REST

Objectives

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

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)
  - PATH 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

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

Code on Demand (optional)

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

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')
```

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.

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

Table 12. 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"

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 verioning 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.

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.

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.

Chapter 18: 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.

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.

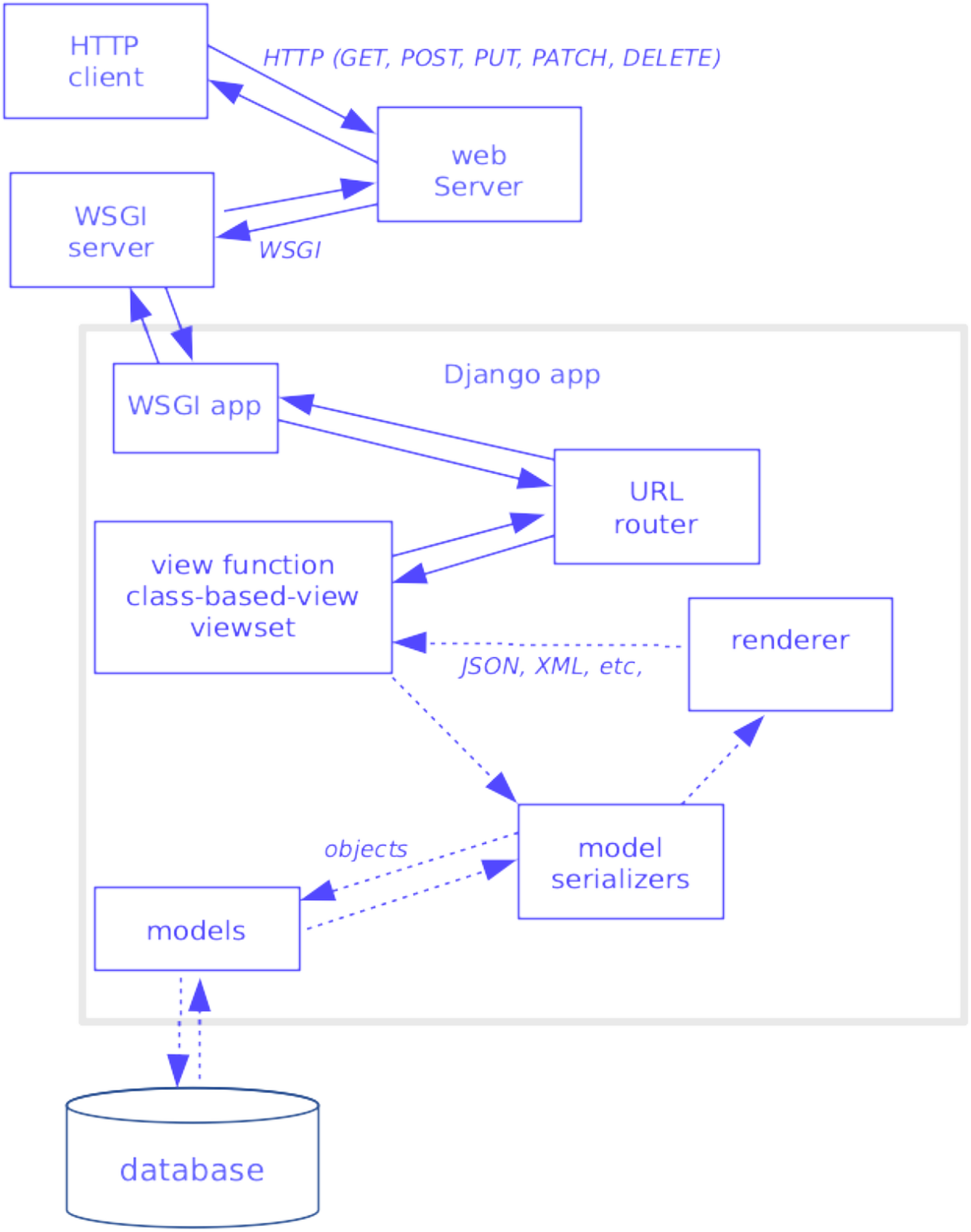


Figure 1. Django REST Architecture

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"}}'
```

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

DJANGO/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')
```

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

DJANGO/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)
```

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

DJANGO/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

DJANGO/contacts/contacts_core/urls.py

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

app_name = 'contacts_core'

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

]
```

Example

DJANGO/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"),
```

Example

DJANGO/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 18 Exercises

Exercise 18-1: (dogs/dogs_core/*)

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

Exercise 18-2: (dogs/dogs_core/*)

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

Chapter 19: Django REST Viewsets

Objectives

- Understand viewsets
- Expose models using viewsets
- Provide pagination

What are viewsets?

- Viewset equals resource
- Higher level of abstraction
- Tools for exposing models
- "Easier than CBVs"

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 ingle 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**.

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 CBVs.

Example

DJANGO/contacts/contacts_core/api/viewsets.py

```
from rest_framework import viewsets
from contacts_core.api.serializers import ContactSerializer, CitySerializer
#  from contacts_core.api.filters import * # (optional)  change to only needed
serializers
from contacts_core.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 13. 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

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

DJANGO/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)),
]
```

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 CBV or viewset if you want to change the pagination scheme for that particular model.

NOTE set `pagination_class=None` to disable pagination.

Chapter 19 Exercises

Exercise 19-1 (dogs/dogs_core/*)

Add viewsets to your dogs app.

(Optional): Add pagination.

Chapter 20: Static file management

Objectives

- Learn what static files are
- Configure locations for static files
- Setting locations outside of projects

Serving static files

- Files that are not modified by server or app
- Passed through to browser
- Several different types
  - CSS
  - JavaScript
  - Images

Static files are files that are not modified by your web application, but are sent to the browser "as-is". Django provides tools to organize and retrieve your static files.

There are several different types of static files. The primary types of static files you need to manage are CSS, JavaScript, and images. There may be others, as well.

Static files do not have to all be served from the same place, although for simple apps this makes things easy.

Configuring access

- Set STATIC_URL in settings.py
- Typically set to /static/
- Put files in …/app/static/app
- Use {% static %} tag and relative path

To set up static files, first assign the value of STATIC_URL in settings.py. The usual location is /static/, which means they will be served from a URL that starts with static/ on your website. The files will actually be served from project/app/static/app.

It is customary to prefix the files with the app name, because all of the static folders in all of the apps in a project will be treated as one single folder.

Setting additional locations

- Set STATICFILES_DIRS to list of folders
- Use forward slashes (even on Windows)
- Will be searched in addition to app/static

There might be a need to serve static files that are outside an app, or even outside the project. For this, you can set the variable STATICFILES_DIRS to a list of one or more full paths; these will be searched in addition to the standard location described previously.

Be sure to use forward slashes for the paths, as they will work fine on Windows, and will not cause any backslash issues.

You can specify a tuple containing a prefix and a path. In this case, the the prefix will be a subfolder of STATIC_URL. Thus, the entry ("images", "/archive/images/web") would let you refer to the file /archive/images/web/monkey.png as /static/images/monkey.png.

Deploying for Production

- Two options
  - Serve from same site
  - Use separate server
- Use manage.py collectstatic

To serve static files in production, first push your code to the deployment server. Then use manage.py to run the collectstatic command. This will collect all static files into STATIC_ROOT.

Configure the web server to serve the files in STATIC_ROOT under the URL in STATIC_URL.

For Apache, this will look something like:

```
<Directory /path/to/mysite.com/static>
Require all granted
</Directory>
```

NOTE You can also serve files from a cloud server (like Amazon S3) or a CDN (content delivery network).

Chapter 20 Exercises

Exercise 20-1 (DJANGO/music)

Download some pictures of your bands. Put the pictures in **static/images folder**, and add them to the band details page.

Chapter 21: 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__.

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**).

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.

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

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.

Example

DJANGO/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"])
```

Chapter 21 Exercises

Exercise 21-1 (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
```

Chapter 22: Django Caching

Objectives

- Understand caching
- Discover cache types
- Set up simple cache
- Explore cache security

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.

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.

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',
    }
}
```

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 14. 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

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)),
]
```

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).

Chapter 23: Reusable Apps

Objectives

- Learn about packages vs apps
- Write setup files
- Create a reusable package
- Distribute and deploy packages

Reusable packages

- Most apps don't break new ground
- Once you implement code, reuse it
- Can be used in multiple projects/apps

Most applications do similar things. The data and the interface make apps unique, but under the hood the mechanics of models, templates, sessions, and so forth are pretty similar from app to app.

Rather than reinventing the wheel for every new project, you can take advantage of installable Django apps that have already been written.

There are plenty of apps on PyPI, and of course you can write your own that are specific to your development needs.

See https://djangopackages.org/ for a repository of reusable Django apps.

See http://racingtadpole.com/blog/reusable-django-apps/ for a nice tutorial on reusable apps

Packages vs Apps

- Package is folder containing related Python files
- App is package written to work with Django

A package is a collection of related Python files that comprise a library or application. Packages are contained in folders, and may have subfolders as needed. It is the highest-level unit of code reuse in Python.

A Django app is a package that has been written to take advantage the Django framework. Thus, a Django app has standard modules such as models.py, views.py, and urls.py (however, these can all be changed by updating configuration files).

A Django project is also a package, but projects are not usually designed for reuse.

A given Django project is almost always a combination of one or more Django apps, plus one or many external apps. Many apps come with Django, and are preconfigured; this is how we get the admin tools, sessions, etc.

How to package

- Copy app to new folder
- Add standard files
- Use setuptools to build package

To create a separate, reusable package, copy the app from the Django project to a new folder.

Create the file **setup.py**.

Use setuptools to build the package as an installable source or binary package.

NOTE See Packaging for the details on generic Python packaging.

Configuring setup.py

- Use **find_packages()**
- Use **install_requires**
- Use `zip_safe = False`

The setup.py script will call the setup() function. In it, there are several things that are useful for Django.

The **find_packages()** function will search your project folder for the packages.

Adding `zip_safe = False` will require your app to be installed as individual files, rather than being zipped up into an **egg** file.

The parameter:

```
install_requires = [
    'django-singleton-admin',
],
```

will help prevent singleton classes from ending up with more than one instance in the Django admin panel.

NOTE Find more information on here: https://pypi.python.org/pypi/django-singleton-admin/0.0.4

What to do next?

- Build package (wheel)
- Install to Python library for sharing
- Share publicly
  - Upload to
    - Github
    - PyPI
  - Add to Django Packages

Now what? Once your project is configured, build a wheel file (*myapp*.whl), and install to a local Python library for sharing (see next section).

NOTE	You will need to install the **wheel** package in order to create wheel files with **setup.py**.

If you want to share publicly, you can upload to github or PyPI. You will want to add it to the Django Packages [https://djangopackages.org/] site.

Installing a package

- Use pip for wheels
- Use setup.py for source

One of the advantages of wheels is that they make installing packages easier. You can just use

```
pip install my_django_app.whl
```

If you have a source distribution, extract the source in any convenient location (this is not permanent) and cd to the top-level folder. Use the following command:

```
python setup.py install
```

To install in the default location.

Use the **--target** option:

```
python setup.py install target=ALTERNATE-DIR
```

to install to an alternate folder, rather than the default location.

NOTE	If you are using a virtual environment (and you should be!), just install the app to the VE as usual.

Adding an app to a Django project

- Make sure app is in PYTHONPATH
- Add to INSTALLED_APPS in setup.py
- Run manage.py migrate if necessary
- Add app's URL config to project URL config

To add the installed app to your Django project, you must do several things:

First, make sure that Django can find the app. This usually means putting the parent folder of the installed app in the environment variable PYTHONPATH. The second is to add the app to the **INSTALLED_APPS** list in your project's setup.py files.

Lastly, be sure to add the app's URL config to the project's URL config, so the views in the app are available.

Chapter 23 Exercises

Exercise 23-1 (DJANGO/music/hello)

Create a new project. In the project, create a minimal "hello, world" app. Package the app using setuptools.

Now add the app to your music project.

Appendix A: Python Bibliography

Title	Author	Publisher
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

Title	Author	Publisher
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.
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, 3nd. 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		

Title	Author	Publisher
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
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

Appendix B: Packaging

How to package

- Copy app to new folder
- Add standard files
- Use setuptools to build package

To create a Python package, add standard files, as described next to the top level of the project.

Create the **setup.py** configuration module (which calls the **setup** function.

Use setuptools to build the package as an installable source or binary package.

Package files

- README.rst reStructuredText setup doc
- MANIFEST.in list of all files in package
- LICENSE s/w license
- setup.py controls packaging and installation

There are four files to create in the package.

README.rst is a typical README file, that describes what the package does in brief. It is not the documentation for the package. It should tell how to install the package. Is is usual to use reStructuredText for this file, hence the .rst extension.

MANIFEST.in is a list of all the non-Python files in the package, such as templates, CSS files, and images. It should also include LICENSE and README.rst.

LICENSE is a file describing the license under which you're releasing the software. This is less important for in-house apps, but essential for apps that are distributed to the public.

The most import file is **setup.py**. This is a Python script that uses setuptools to control how your application is packaged for distribution, and how it is installed by users.

setup.py contains a call to the setup() function; named options configure the details of your package.

TIP The following link has some great suggestions for laying out the files in a package: https://blog.ionelmc.ro/2014/05/25/python-packaging/

Overview of setuptools

- Creates distributable files
- Can be used for modules or packages
- Many distribution formats
- Configuration in setup.py

The key to using setuptools is setup.py, the configuration file. This tells setuptools what files should go in the module, the version of the application or package, and any other configuration data.

The steps for using setuptools are:

- write a setup script (setup.py by convention)
- (optional) write a setup configuration file
- create a source, wheel, or specialized built distributions

The entire process is described at https://packaging.python.org/distributing/.

See https://packaging.python.org/glossary/#term-built-distribution for a glossary of packaging and distribution terms.

Preparing for distribution

- Organize files and folders
- Create setup.py

The first step is to create setup.py in the package folder.

setup.py is a python script that calls the setup() function from setuptools with keyword (named) arguments that describe your module.

For modules, use the py_modules keyword; for packages, use the packages.

There are many other options for fine-tuning the distribution.

Your distribution should also have a file named README or README.txt which can be a brief description of the distribution and how to install its module(s).

You can include any other files desired. Many developers include a LICENSE.txt which stipulates how the distribution is licensed.

Table 15. Keyword arguments for setup() function

Keyword	Description
author	Package author's name
author_email	Email address of the package author
classifiers	A list of classifiers to describe level (alpha, beta, release) or supported versions of Python
data_files	Non-Python files needed by distribution from outside the package
description	Short, summary description of the package
download_url	Location where the package may be downloaded
entry_points	Plugins or scripts provided in the package. Use key console_scripts to provide standalone scripts.
ext_modules	Extension modules needing special handling
ext_package	Package containing extension modules
install_requires	Dependencies. Specify modules your package depends on.
keywords	Keywords that describe the project
license	license for the package
long_description	Longer description of the package
maintainer	Package maintainer's name
maintainer_email	Email address of the package maintainer
name	Name of the package
package_data	Additional non-Python files needed from within the package
package_dir	Dictionary mapping packages to folders
packages	List of packages in distribution
platforms	A list of platforms
py_modules	List of individual modules in distribution
scripts	Configuration for standalone scripts provided in the package (but entry_points is preferred)
url	Home page for the package
version	Version of this release

Creating a source distribution

- Use setup.py with -sdist option
- Creates a platform-neutral distribution
- Distribution has its own setup.py

Run setup.py with your version of python, specifying the -sdist option. This will create a platform-independent source distribution. python setup.py sdist

```
ls -l dist
total 4
-rw-rw-r-- 1 jstrick jstrick 633 2012-01-11 07:49 temperature-1.2.tar.gz

tar tzvf dist/temperature-1.2.tar.gz
drwxrwxr-x jstrick/jstrick   0 2012-01-11 00:26 temperature-1.2/
-rw-rw-r-- jstrick/jstrick 256 2012-01-11 00:26 temperature-1.2/PKG-INFO
-rw-rw-r-- jstrick/jstrick 285 2012-01-10 13:36 temperature-1.2/setup.py
-rw-r--r-- jstrick/jstrick 342 2012-01-10 07:52 temperature-1.2/temperature.py
```

To install a source distribution, extract it into any directory and cd into the root (top) of the extracted file structure. Execute the following command:

```
python setup.py install
```

Example

temperature/setup.py

```
from setuptools import setup
# from setuptools import find_packages

setup(
    name='temperature',
    version='1.2',
    description='Class for converting temperatures',
    long_description='Long description here....',
    author='John Strickler',
    author_email='jstrickler@gmail.com',
    license='MIT',
    url='http://www.cja-tech.com/temperature',  # doesn't really exist
    py_modules=['temperature'],
#    packages=find_packages(exclude=['contrib', 'doc', 'tests']),
)
```

```
cd temperature
python setup.py sdist
running sdist
running egg_info
writing temperature.egg-info/PKG-INFO
writing top-level names to temperature.egg-info/top_level.txt
writing dependency_links to temperature.egg-info/dependency_links.txt
reading manifest file 'temperature.egg-info/SOURCES.txt'
writing manifest file 'temperature.egg-info/SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.rst,
README.txt

running check
creating temperature-1.0.0
creating temperature-1.0.0/temperature.egg-info
making hard links in temperature-1.0.0...
hard linking setup.py -> temperature-1.0.0
hard linking temperature.py -> temperature-1.0.0
hard linking temperature.egg-info/PKG-INFO -> temperature-1.0.0/temperature.egg-info
hard linking temperature.egg-info/SOURCES.txt -> temperature-1.0.0/temperature.egg-info
hard linking temperature.egg-info/dependency_links.txt -> temperature-
1.0.0/temperature.egg-info
hard linking temperature.egg-info/top_level.txt -> temperature-1.0.0/temperature.egg-info
Writing temperature-1.0.0/setup.cfg
Creating tar archive
removing 'temperature-1.0.0' (and everything under it)
```

```
tree dist
dist
└──── temperature-1.2.tar.gz
```

Creating wheels

- 3 kinds of wheels
  - Universal wheels (pure Python; python 2 *and* 3 compatible
  - Pure Python wheels (pure Python; Python 2 *or* 3 compatible
  - Platform wheels (Platform-specific; binary)

A wheel is prebuilt distribution. Wheels can be installed with pip.

A Universal wheel is a pure Python package (no extensions) that can be installed on either Python 2 or Python 3. It has to have been carefully written that way.

A Pure Python wheel is a pure Python package that is specific to one version of Python (either 2 or 3). It can only be installed by a matching version of pip.

A Platform wheel is a package that has extensions, and thus is platform-specific.

Example

```
python setup.py bdist_wheel
running bdist_wheel
running build
running build_py
creating build
creating build/lib
copying temperature.py -> build/lib
installing to build/bdist.macosx-10.6-x86_64/wheel
running install
running install_lib
creating build/bdist.macosx-10.6-x86_64
creating build/bdist.macosx-10.6-x86_64/wheel
copying build/lib/temperature.py -> build/bdist.macosx-10.6-x86_64/wheel
running install_egg_info
running egg_info
writing temperature.egg-info/PKG-INFO
writing top-level names to temperature.egg-info/top_level.txt
writing dependency_links to temperature.egg-info/dependency_links.txt
reading manifest file 'temperature.egg-info/SOURCES.txt'
writing manifest file 'temperature.egg-info/SOURCES.txt'
Copying temperature.egg-info to build/bdist.macosx-10.6-x86_64/wheel/temperature-1.0.0-
py2.7.egg-info
running install_scripts
creating build/bdist.macosx-10.6-x86_64/wheel/temperature-1.0.0.dist-info/WHEEL
```

```
tree dist
dist
├──── temperature-1.2-py3-none-any.whl
└──── temperature-1.2.tar.gz
```

Creating other built distributions

- Use setup.py with -bdist –format=format
- Creates platform-specific distributions
- Common Unix formats: rpm, deb, tgz
- Common Windows formats: msi, exe

For the convenience of the end-user, you can create "built" distributions, which are ready to install on specific platforms. These are built with the bdist argument to setup.py, plus a --format=format option to indicate the target platform.

```
python setup.py bdist --format=rpm
```

```
tree dist
dist
├──── temperature-1.2-py3-none-any.whl
├──── temperature-1.2.macosx-10.9-x86_64.tar
├──── temperature-1.2.macosx-10.9-x86_64.zip
└──── temperature-1.2.tar.gz
```

Using Cookiecutter

- Create standard layout
- Developed for Django
- Very flexible

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 many templates on **github** to choose from, and you can easily create your own.

The script 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.

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

Installing a package

- Use pip for wheels
- Use setup.py for source

One of the advantages of wheels is that they make installing packages easier. You can just use

pip install *package*.whl

If you have a source distribution, extract the source in any convenient location (this is not permanent) and cd to the top-level folder. Use the following command:

```
python setup.py install
```

To install in the default location.

Use

```
python setup.py install --prefix=ALTERNATE-DIR
```

to install under an alternate prefix.

Index

