



Quick answers to common problems

Web Development with Django Cookbook

Over 70 practical recipes to create multilingual, responsive, and scalable websites with Django

Aidas Bendoraitis

[PACKT] open source 
PUBLISHING community experience distilled

Web Development with Django Cookbook

Over 70 practical recipes to create multilingual,
responsive, and scalable websites with Django

Aidas Bendoraitis

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Web Development with Django Cookbook

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2014

Production reference: 1091014

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-689-8

www.packtpub.com

Cover image by David Puerta (dpuerta.herrero@gmail.com)

Credits

Author

Aidas Bendoraitis

Project Coordinator

Harshal Ved

Reviewers

Thiago Carvalho D'Ávila

Michael Giuliano

Yuxian Eugene Liang

Danijel Pančić

Proofreaders

Simran Bhogal

Maria Gould

Ameesha Green

Paul Hindle

Commissioning Editor

Greg Wild

Indexers

Monica Ajmera Mehta

Priya Sane

Acquisition Editor

Greg Wild

Production Coordinator

Adonia Jones

Content Development Editor

Athira Laji

Cover Work

Adonia Jones

Technical Editor

Tanvi Bhatt

Copy Editors

Simran Bhogal

Janbal Dharmaraj

Maria Gould

Paul Hindle

Sayanee Mukherjee

About the Author

Aidas Bendoraitis has been professionally working with web technologies for over a decade. Over the last 8 years at a Berlin-based company, studio 38 pure communication GmbH, together with the creative team, he has developed a number of small- and large-scale Django projects, mostly in the cultural sector. At the moment, he is also working as a software architect at a London-based mobile start-up, Hype.

About the Reviewers

Thiago Carvalho D'Ávila graduated with a degree in Information Systems and has studied the Architecture of Distributed Systems. He has worked with various software platforms including Python (with Django/ Web2PY), .NET, SAP IS-U and CRM modules, Java, and Arduino.

Thiago has been using Django since release 1.4, testing beta versions and always trying the new features and concepts included in the framework. He has developed a couple of open source apps for responsive design using Bootstrap.

Michael Giuliano has been developing software in various languages and technologies for the past 15 years. Having used Python in the fields of web services, machine learning, and big data since 2008, he finds it to be one of the most versatile, elegant, and productive programming languages.

Michael is currently based in London, where he leads the Python development team at Zoopla Property Group Plc.

Yuxian Eugene Liang is a researcher, author, web developer, and business developer. He has experience in both frontend and backend development, in particular, engineering user experiences using JavaScript/CSS/HTML and performing social network analysis. He most recently led a team of two (including himself) to be the champions at Startup Weekend, where he played the role of a team leader, designer, and frontend engineer. He is currently interested in engineering user experiences and growth hacking. His previous books include *JavaScript Testing Beginners Guide* and *Wordpress Mobile Applications with PhoneGap*, both published by Packt Publishing. He can be found at <http://www.liangeugene.com>.

Danijel Pančić is a JavaScript ninja and passionate Django enthusiast. For the past 5 years, he's been working at Zemanta as a developer on projects of all scales, ranging from tools that made life easier to finished products that were published online for thousands of users. He also works on various projects, including online games, in his spare time and experiments with new approaches and techniques in search of better ways to achieve results.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Getting Started with Django 1.6	7
Introduction	8
Working with a virtual environment	8
Creating a project file structure	10
Handling project dependencies with pip	13
Including external dependencies in your project	14
Defining relative paths in the settings	17
Setting up STATIC_URL dynamically for Subversion users	18
Setting up STATIC_URL dynamically for Git users	19
Creating and including local settings	21
Setting UTF-8 as the default encoding for MySQL configuration	22
Setting the Subversion ignore property	23
Creating the Git ignore file	25
Deleting Python-compiled files	26
Importing an order in Python files	26
Defining overwritable app settings	28
Chapter 2: Database Structure	31
Introduction	31
Using model mixins	32
Creating a model mixin with URL-related methods	33
Creating a model mixin to handle creation and modification dates	36
Creating a model mixin to take care of meta tags	38
Creating a model mixin to handle generic relations	41
Handling multilingual fields	45
Using South migrations	52
Changing a foreign key to the many-to-many field with South	54

Chapter 3: Forms and Views	57
Introduction	57
Passing HttpRequest to the form	58
Utilizing the save method of the form	60
Uploading images	62
Creating a form layout with django-crispy-forms	67
Filtering object lists	72
Managing paginated lists	79
Composing class-based views	82
Generating PDF documents	85
Chapter 4: Templates and JavaScript	93
Introduction	93
Arranging the base.html template	94
Including JavaScript settings	97
Using HTML5 data attributes	100
Opening object details in a pop up	104
Implementing a continuous scroll	108
Implementing the Like widget	110
Uploading images by Ajax	118
Chapter 5: Custom Template Filters and Tags	127
Introduction	127
Following conventions for your own template filters and tags	128
Creating a template filter to show how many days have passed	129
Creating a template filter to extract the first media object	131
Creating a template filter to humanize URLs	133
Creating a template tag to include a template if it exists	134
Creating a template tag to load a QuerySet in a template	137
Creating a template tag to parse content as a template	141
Creating a template tag to modify request query parameters	143
Chapter 6: Model Administration	147
Introduction	147
Customizing columns in the change list page	147
Creating admin actions	151
Developing change list filters	155
Exchanging administration settings for external apps	158
Inserting a map into a change form	160

Chapter 7: Django CMS	169
Introduction	169
Creating templates for Django CMS	170
Structuring the page menu	174
Converting an app to a CMS app	177
Attaching your own navigation	179
Writing your own CMS plugin	181
Adding new fields to the CMS page	187
Chapter 8: Hierarchical Structures	195
Introduction	195
Creating hierarchical categories	197
Creating a category administration interface with django-mptt-admin	200
Creating a category administration interface with django-mptt-tree-editor	202
Rendering categories in a template	204
Using a single selection field to choose a category in forms	206
Using a checkbox list to choose multiple categories in forms	208
Chapter 9: Data Import and Export	213
Introduction	213
Importing data from a local CSV file	213
Importing data from a local Excel file	215
Importing data from an external JSON file	217
Importing data from an external XML file	222
Creating filterable RSS feeds	227
Using Tastypie to provide data to third parties	232
Chapter 10: Bells and Whistles	237
Introduction	237
Using the Django shell	238
The monkey patching slugification function	240
The monkey patching model administration	242
Toggling Debug Toolbar	247
Using ThreadLocalMiddleware	251
Caching the method value	253
Getting detailed error reporting via e-mail	254
Deploying on Apache with mod_wsgi	257
Creating and using the Fabric deployment script	264
Index	275

Preface

The Django framework is relatively easy to learn and it solves many web-related questions such as project structure, database object-relational mapping, templating, form validation, sessions, authentication, security, cookies, internationalization, basic administration, creating interfaces to access data from scripts, and more. Django is based on the Python programming language, whose code is clear and easy to read. Django also has a lot of third-party modules that can be used in conjunction with your own apps. Django has an established and vibrant community where you can find source code, get help, and contribute.

Django Web Development Cookbook will guide you through all web development processes with the Django 1.6 framework. You will get started with the virtual environment and configuration of the project and then you will learn how to define a database structure with reusable components. After this, you will move on to exploring the forms and views used to enter and list data. Then, you will continue with responsive templates and JavaScript to create the best user experience. After that, you will find out how to tweak the administration to make the website editors happy. You will also learn how to integrate your own functionality into Django CMS. The next step will be learning how to use hierarchical structures. Then, you will find out that collecting data from different sources and providing data to others in different formats isn't as difficult as you thought. And finally, you'll be introduced to some programming and debugging tricks and will be shown how to deploy the project to a remote dedicated server.

In contrast to other Django books, this book will deal not only with the code of the framework itself, but also with some important third-party modules necessary for fully equipped web development. The book also gives examples of rich user interfaces using the Bootstrap frontend framework and the jQuery JavaScript library.

What this book covers

Chapter 1, Getting Started with Django 1.6, will guide you through the basic configuration that is necessary to start any Django project. It will cover topics such as the virtual environment, version control, and project settings.

Chapter 2, Database Structure, will teach you how to write reusable pieces of code to use in your models. When you create a new app, the first thing to do is to define your models. Also, you will be told how to manage database schema changes using South migrations.

Chapter 3, Forms and Views, will show you some patterns used to create the views and forms for your data.

Chapter 4, Templates and JavaScript, will show you practical examples of using templates and JavaScript together. We bring together templates and JavaScript because information is always presented to the user by rendered templates and in modern websites, JavaScript is a must for a richer user experience.

Chapter 5, Custom Template Filters and Tags, will show you how to create and use your own template filters and tags, as the default Django template system is quite extensive, and there are more things to add for different cases.

Chapter 6, Model Administration, will guide you through extending the default administration with your own functionality, as the Django framework comes with a handy prebuilt model administration.

Chapter 7, Django CMS, deals with the best practices of using Django CMS, which is the most popular open source content management system made with Django, and extending it for your needs.

Chapter 8, Hierarchical Structures, shows that whenever you need to create a tree-like structure in Django, the `django-mptt` module comes in handy. This chapter shows you how to use it and how to set administration for hierarchical structures.

Chapter 9, Data Import and Export, demonstrates to us that very often there are cases when we need to transfer data from and to different formats, and retrieve it from and provide it to different sources. This chapter deals with management commands for data import and also APIs for data export.

Chapter 10, Bells and Whistles, will show some additional snippets and tricks useful in web development, debugging, and deployment.

What you need for this book

To develop with Django 1.6, you will need Python 2.6 or 2.7, the Pillow library for image manipulation, the MySQL database and MySQLdb bindings or PostgreSQL, virtualenv to keep each project's Python modules separated, and Git or Subversion for version control.

All other specific requirements are mentioned in each recipe separately.

Who this book is for

If you have created websites with Django but you want to sharpen your knowledge and learn some good approaches for how to treat different aspects of web development, this book is for you. It is intended for intermediate and professional Django users who need to build projects that must be multilingual, functional on devices of different screen sizes, and which scale over time.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the `models.py` file of the `guerrilla_patches` app, add the following content."

A block of code is set as follows:

```
# -*- coding: UTF-8 -*-
from django.utils import text
from slugify import slugify_de as awesome_slugify
awesome_slugify.to_lower = True
text.slugify = awesome_slugify
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

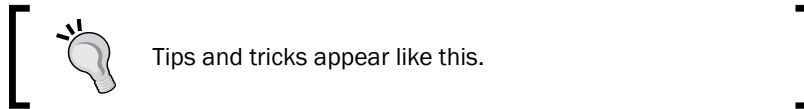
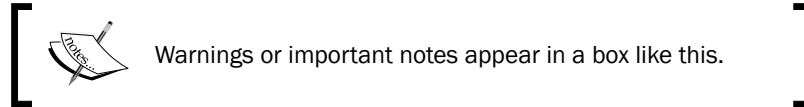
```
class ExtendedChangeList(main.ChangeList):

    def get_translated_list(self, model, field_list):
        language = get_language()
        translated_list = []
        opts = model._meta
```

Any command-line input or output is written as follows:

```
(myproject_env)$ fab dev deploy
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Go to **Webmin** | **System** | **Scheduled Cron Jobs** | **Create a new scheduled cron job** and create a scheduled cron job with these properties."



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started with Django 1.6

In this chapter, we will cover the following topics:

- ▶ Working with a virtual environment
- ▶ Creating a project file structure
- ▶ Handling project dependencies with pip
- ▶ Including external dependencies in your project
- ▶ Defining relative paths in the settings
- ▶ Setting up STATIC_URL dynamically for Subversion users
- ▶ Setting up STATIC_URL dynamically for Git users
- ▶ Creating and including local settings
- ▶ Setting UTF-8 as the default encoding for MySQL configuration
- ▶ Setting the Subversion ignore property
- ▶ Creating a Git ignore file
- ▶ Deleting Python-compiled files
- ▶ Importing order in Python files
- ▶ Defining overwritable app settings

Introduction

In this chapter, I will show you good practices when starting a new project with Django 1.6 on Python 2.7. Some of the tricks introduced here are the best ways to deal with the project layout, settings, or configuration. However, for some of the tricks, you might find some alternatives online. So, feel free to evaluate and choose the best bits and pieces for yourself while digging deep into the Django world.

I am assuming that you are already familiar with the basics of Django, Subversion or Git version control, MySQL or PostgreSQL databases, and command-line usage. Also, I assume you are probably using a Unix-based operating system such as Mac OS X or Linux. It makes sense to develop with Django on Unix-based platforms because the websites will most likely be published on a Linux Server later, so you can establish routines that work the same while developing as while deploying. If you are working with Django locally on Windows, the routines are similar, but not always exactly the same.

Working with a virtual environment

It is very likely that you will develop multiple Django projects on your computer. Some modules, such as **Python Image Library** (or **Pillow**) and **MySQLdb**, can be installed once and shared for all projects. Other modules such as Django itself, third-party Python libraries, and Django apps will need to be kept isolated from each other. **Virtualenv** is a utility that separates all your Python projects into their own realms. In this recipe, we will show you how to use it.

Getting ready

Before getting into the usage example of virtual environments, let's install pip (the most convenient tool to install and manage Python packages), the shared Python modules Pillow and MySQLdb, and the virtualenv utility using the following commands:

```
$ sudo easy_install pip
$ sudo pip install Pillow
$ sudo pip install MySQL-python
$ sudo pip install virtualenv
```

How to do it...

Now, when you have your prerequisites installed, create a directory where all your Django projects will be stored, for example, `virtualenvs` under your home directory. Perform the following steps after creating the directory:

1. Go to the newly created directory and create a virtual environment that uses shared system site packages:

```
$ cd ~/virtualenvs
$ mkdir myproject_env
$ cd myproject_env
$ virtualenv --system-site-packages .
New python executable in ./bin/python
Installing setuptools.....done.
Installing pip.....done.
```
2. To use your newly created virtual environment, you need to execute the activation script within your current shell. This can be done with one of the following commands:

```
$ source bin/activate
$ . bin/activate
```
3. You will see that the prompt of the command-line tool gets a prefix of the project name, such as this:

```
(myproject_env)$
```
4. To get out of the virtual environment, type the following command:

```
$ deactivate
```

How it works...

When you create a virtual environment, specific directories (`bin`, `build`, `include`, and `lib`) are created to store a copy of the Python installation, and some shared Python paths are defined. When the virtual environment is activated, whatever you install with `pip` or `easy_install` will be put into and used by the site packages of the virtual environment, and not the global site-packages of your Python installation.

To install Django 1.6 to your virtual environment, type the following command:

```
(myproject_env)$ pip install Django==1.6
```

See also

- ▶ The *Creating a project file structure* recipe
- ▶ The *Deploying on Apache with mod_wsgi* recipe in *Chapter 10, Bells and Whistles*

Creating a project file structure

A consistent file structure for your projects makes you well organized and more productive. When you have the basic workflow defined, you can get into the business logic quicker and create awesome projects.

Getting ready

If you haven't done this yet, first create the `virtualenvs` directory where you will keep all your virtual environments (read about this in the *Working with a virtual environment* recipe), and which can be created under your home directory.

Then, create the directory for your project's environment, for example, `myproject_env`. Start the virtual environment inside it. This will create the `bin`, `build`, `include`, and `lib` directories there. I suggest adding the `commands` directory for local bash scripts related to the project, the `db_backups` directory for database dumps, and the `project` directory for your Django project. Also, install Django into your virtual environment.

How to do it...

Follow these steps to create a file structure for your project:

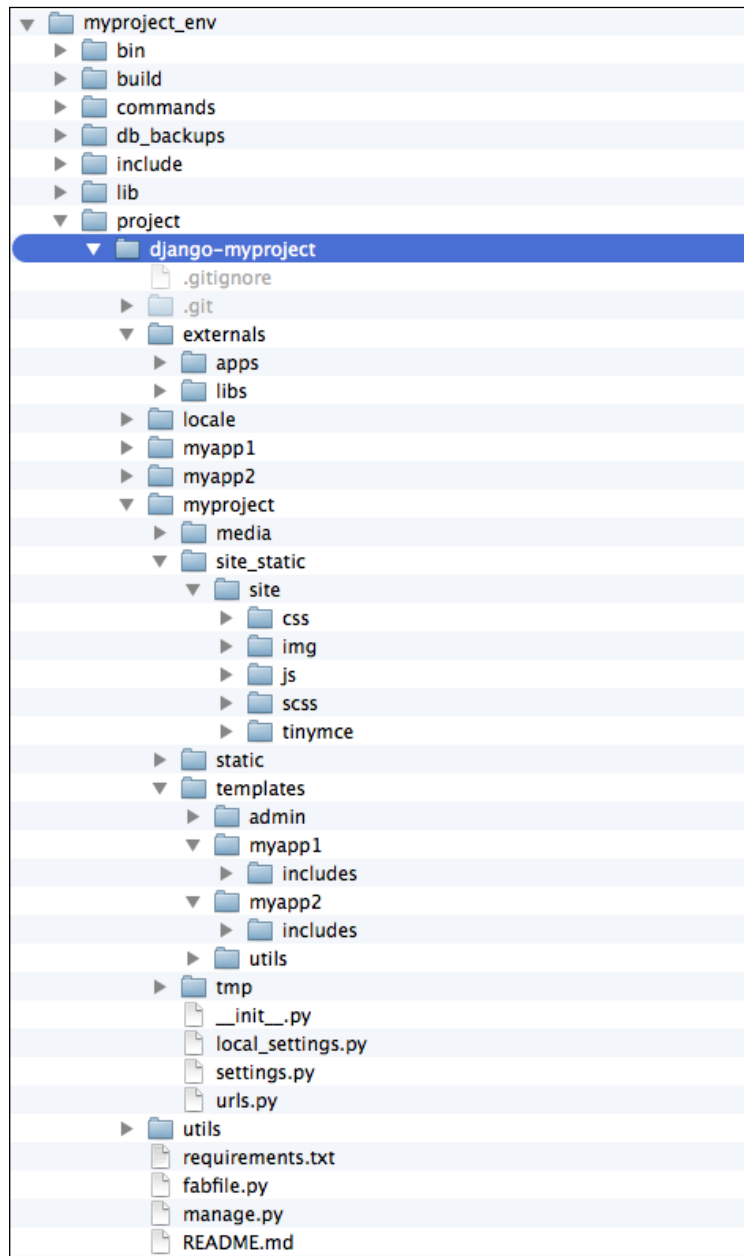
1. With the virtual environment activated, go to the `project` directory and start a new Django project, as follows:

```
(myproject_env)$ django-admin.py startproject myproject
```
2. For clearness, we will rename the newly created directory to `django-myproject`. This is the directory that you should put under version control, so it will have `.git`, `.svn`, or similar directories inside.

3. In the `django-myproject` directory, create a `README.md` file to describe your project to new developers. You can also put `requirements.txt` with the Django version, and you can include other external dependencies (read about this in the *Handling project dependencies with pip* recipe). Also, this directory will contain your project's Python package named `myproject`, Django apps (I recommend to have one app called `utils` for different functionalities shared throughout the project), a locale directory for your project translations if it is multilingual, a Fabric deployment script named `fabfile.py`, and the `externals` directory for external dependencies included in this project if you decide not to use `requirements.txt`.
4. In your project's Python package, `myproject`, create the `media` directory for project uploads, the `site_static` directory for project-specific static files, the `static` directory for collected static files, the `tmp` directory for the upload procedure, and the `templates` directory for project templates. Also, the `myproject` directory should contain your project settings, `settings.py` and `local_settings.py`, as well as the URL configuration, `urls.py`.
5. In your `site_static` directory, create the `site` directory as a namespace for site-specific static files. Then, put the static files separated into directories inside it; for instance, `scss` for Sass files (optional), `css` for generated minified cascading style sheets, `img` for styling images and logos, `js` for JavaScript, and lastly, any third-party module combining all types of files, for example, the rich text editor `tinymce`. Besides the `site` directory, the `site_static` directory might also contain overwritten static directories of third-party apps, for example, `cms` overwriting static files from Django CMS. To generate the CSS files out of Sass and to minify your JavaScript files, you can use the CodeKit or Prepros applications with a graphical user interface.
6. Put your templates that are separated by apps into your `templates` directory. If a template file represents a page (for example, `change_item.html` or `item_list.html`), then put it directly in the app's `template` directory. If the template is included in another template (for example, `similar_items.html`), put it into the `includes` subdirectory. Also, your `templates` directory can contain one directory called `utils` for globally reusable snippets such as pagination, language chooser, and others.

How it works...

The whole file structure for a complete project inside a virtual environment will look like this:



See also

- ▶ The *Handling project dependencies with pip* recipe
- ▶ The *Including external dependencies in your project* recipe
- ▶ The *Deploying on Apache with mod_wsgi* recipe in *Chapter 10, Bells and Whistles*
- ▶ The *Creating and using the Fabric deployment script* recipe in *Chapter 10, Bells and Whistles*

Handling project dependencies with pip

Pip is the most convenient tool to install and manage Python packages. Besides installing packages one by one, it is possible to define a list of packages you want to install and pass it to the tool so that it deals with the list automatically.

Getting ready

Before using this recipe, you need to have pip installed and a virtual environment activated. For more information on how to do this, read the *Working with a virtual environment* recipe.

How to do it...

Let's go to your Django project that you have under version control and create the requirements file with the following content:

```
#requirements.txt
Django==1.6
South==0.8.4
django-cms==2.4
```

Now, you can run the following command to install all required dependencies for your Django project:

```
(myproject_env)$ pip install -r requirements.txt
```

How it works...

This command installs all your project dependencies into your virtual environment one after another.

When you have many dependencies in your project, it is good practice to stick to specific versions of the Python modules because you can then be sure that when you deploy your project or give it to a new developer, the integrity doesn't get broken and all the modules function without conflicts.

If you have already installed project requirements with pip manually one by one, you can generate the `requirements.txt` file using the following command:

```
(myproject_env)$ pip freeze > requirements.txt
```

There's more...

If you need to install a Python library directly from a version control system or a local path, you can learn more about pip from the official documentation at http://pip.readthedocs.org/en/latest/reference/pip_install.html.

See also

- ▶ The *Working with a virtual environment* recipe
- ▶ The *Including external dependencies in your project* recipe

Including external dependencies in your project

Sometimes, it is better to include external dependencies in your project. This ensures that whenever one developer upgrades third-party modules, all the other developers will receive the upgraded version within the next update from the version control system (Git, Subversion, or others).

Also, it is good to have external dependencies included in your project when the libraries are taken from unofficial sources (somewhere other than **Python Package Index (PyPI)**) or different version control systems.

Getting ready

Start with a virtual environment with a Django project in it.



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

How to do it...

Execute the following steps one by one:

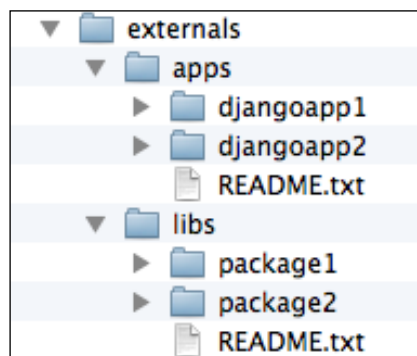
1. Create a directory named `externals` under your Django project.
2. Then, create the `libs` and `apps` directories under it.

The `libs` directory is for Python modules that are required by your project, for example, `requests`, `boto`, `twython`, `whoosh`, and so on. The `apps` directory is for third-party Django apps, for example, `django-haystack`, `django-cms`, `django-south`, `django-storages`, and so on.



I highly recommend that you create `README.txt` files inside the `libs` and `apps` directories, where you mention what each module is for, what the used version or revision is, and where it is taken from.

3. So, the directory structure should look like this:



4. The next step is to put the external libraries and apps under the Python path so that they are recognized as if they were installed. This can be done by adding the following code in the settings:

```
#settings.py
# -*- coding: UTF-8 -*-
import os
import sys

PROJECT_PATH = os.path.abspath(os.path.join(
    os.path.dirname(__file__), ".."))
```

```
EXTERNAL_LIBS_PATH = os.path.join(PROJECT_PATH, "externals",
                                   "libs")
EXTERNAL_APPS_PATH = os.path.join(PROJECT_PATH, "externals",
                                   "apps")
sys.path = ["", EXTERNAL_LIBS_PATH, EXTERNAL_APPS_PATH] + sys.path
```

How it works...

A module is meant to be under the Python path if you can run Python and import the module. One of the ways to put a module under the Python path is to modify the `sys.path` variable before importing a module that is in an unusual location. The value of `sys.path` is a list of directories starting with an empty string for the current directory, followed by the directories in the virtual environment, and finally the globally shared directories of the Python installation. You can see the value of `sys.path` in the Python shell, as follows:

```
(myproject_env)$ python
>>> import sys
>>> sys.path
```

When trying to import a module, Python searches for the module in this list and returns the first result found.

So, at first, we define the `PROJECT_PATH` variable that is the absolute path to one level higher than the `settings.py` file. Then, we define the variables `EXTERNAL_LIBS_PATH` and `EXTERNAL_APPS_PATH`, which are relative to `PROJECT_PATH`. Lastly, we modify the `sys.path` property, adding new paths to the beginning of the list. Note that we also add an empty string as the first path to search, which means that the current directory of any module should always be checked first before checking other Python paths.



This way of including external libraries doesn't work cross-platform with Python packages that have C language bindings, for example, `lxml`. For such dependencies, I recommend using pip's `requirements.txt` file that was introduced in the previous recipe.

See also

- ▶ The *Creating a project file structure* recipe
- ▶ The *Handling project dependencies with pip* recipe
- ▶ The *Defining relative paths in the settings* recipe
- ▶ The *Using the Django shell* recipe in *Chapter 10, Bells and Whistles*

Defining relative paths in the settings

Django requires you to define different filepaths in the settings, such as the root of your media, the root of your static files, the path to templates, the path to translation files, and so on. For each developer of your project, the paths might differ as the virtual environment can be set up anywhere, and the user might be working on Mac OS X, Linux, or Windows. Anyway, there is a way to define these paths relative to your Django project directory.

Getting ready

To start with, open up `settings.py`.

How to do it...

Modify your path-related settings accordingly, instead of hardcoding paths to your local directories:

```
#settings.py
# -*- coding: UTF-8 -*-
import os

PROJECT_PATH = os.path.abspath(os.path.join(os.path.dirname(__file__),
    ".."))

MEDIA_ROOT = os.path.join(PROJECT_PATH, "myproject", "media")

STATIC_ROOT = os.path.join(PROJECT_PATH, "myproject", "static")

STATICFILES_DIRS = (
    os.path.join(PROJECT_PATH, "myproject", "site_static"),
)

TEMPLATE_DIRS = (
    os.path.join(PROJECT_PATH, "myproject", "templates"),
)

LOCALE_PATHS = (
    os.path.join(PROJECT_PATH, "locale"),
)

FILE_UPLOAD_TEMP_DIR = os.path.join(PROJECT_PATH, "myproject", "tmp")
```

How it works...

At first, we define `PROJECT_PATH`, which is an absolute path to one level higher than the `settings.py` file. Then, we set all the paths relative to `PROJECT_PATH` using the `os.path.join` function.

See also

- ▶ The *Including external dependencies in your project* recipe

Setting up `STATIC_URL` dynamically for Subversion users

If you set `STATIC_URL` to a static value, then each time you update a CSS file, a JavaScript file, or an image, you will need to clear the browser cache to see the changes. There is a trick to work around the clearing of the browser's cache. It is to have the revision number of the version control system shown in `STATIC_URL`. Whenever the code is updated, the visitor's browser will force the loading of all-new static files.

This recipe shows how to put a revision number into `STATIC_URL` for Subversion users.

Getting ready

Make sure that your project is under the Subversion version control and you have `PROJECT_PATH` defined in your settings, as shown in the *Defining relative paths in the settings* recipe.

Then, create the `utils` module in your Django project, and also create a file called `misc.py` there.

How to do it...

The procedure for putting the revision number into the `STATIC_URL` setting consists of the following two steps:

1. Insert the following content:

```
#utils/misc.py
# -*- coding: UTF-8 -*-
import subprocess

def get_media_svn_revision(absolute_path):
    repo_dir = absolute_path
```



```

svn_revision = subprocess.Popen(
    'svn info | grep "Revision" | awk \'{print $2}\'',
    stdout=subprocess.PIPE, stderr=subprocess.PIPE,
    shell=True, cwd=repo_dir, universal_newlines=True)
rev = svn_revision.communicate()[0].partition('\n')[0]
return rev

```

2. Then, modify the `settings.py` file and add these lines:

```

#settings.py
# ... somewhere after PROJECT_PATH definition ...
from utils.misc import get_media_svn_revision
STATIC_URL = "/static/%s/" % get_media_svn_revision(PROJECT_PATH)

```

How it works...

The `get_media_svn_revision` function takes the `absolute_path` directory as a parameter and calls the `svn info` shell command in that directory to find out the current revision. We pass `PROJECT_PATH` to the function because we are sure it is under version control. Then, the revision is parsed, returned, and included in the `STATIC_URL` definition.

See also

- ▶ The *Setting up `STATIC_URL` dynamically for Git users* recipe
- ▶ The *Setting the Subversion ignore property* recipe

Setting up `STATIC_URL` dynamically for Git users

If you don't want to refresh the browser cache each time you change your CSS and JavaScript files, or while styling images, you need to set `STATIC_URL` dynamically with a varying path component. With the dynamically changing URL, whenever the code is updated, the visitor's browser will force the loading of all-new uncached static files. In this recipe, we will set a dynamic path for `STATIC_URL` when you use the Git version control system.

Getting ready

Make sure that your project is under the Git version control and you have `PROJECT_PATH` defined in your settings, as shown in the *Defining relative paths in the settings* recipe.

If you haven't done it yet, create the `utils` module in your Django project, and create a file called `misc.py` there.

How to do it...

The procedure for putting the Git timestamp into the `STATIC_URL` setting consists of the following two steps:

1. Add the following content to the `misc.py` file placed at `utils/`:

```
#utils/misc.py
# -*- coding: UTF-8 -*-
import subprocess
from datetime import datetime

def get_git_changeset(absolute_path):
    repo_dir = absolute_path
    git_show = subprocess.Popen(
        'git show --pretty=format:%ct --quiet HEAD',
        stdout=subprocess.PIPE, stderr=subprocess.PIPE,
        shell=True, cwd=repo_dir, universal_newlines=True,
    )
    timestamp = git_show.communicate()[0].partition('\n')[0]
    try:
        timestamp = datetime.utcfromtimestamp(int(timestamp))
    except ValueError:
        return ""
    changeset = timestamp.strftime('%Y%m%d%H%M%S')
    return changeset
```

2. Then, import the newly created `get_git_changeset` function in the settings and use it for the `STATIC_URL` path:

```
#settings.py
# ... somewhere after PROJECT_PATH definition ...
from utils.misc import get_git_changeset
STATIC_URL = "/static/%s/" % get_git_changeset(PROJECT_PATH)
```

How it works...

The `get_git_changeset` function takes the `absolute_path` directory as a parameter and calls the `git show` shell command with the parameters to show the Unix timestamp of the HEAD revision in the directory. As in the previous recipe, we pass `PROJECT_PATH` to the function because we are sure it is under version control. The timestamp is parsed; converted to a string consisting of year, month, day, hour, minutes, and seconds; returned; and included in the definition of `STATIC_URL`.

See also

- ▶ The *Setting up STATIC_URL dynamically for Subversion users* recipe
- ▶ The *Creating the Git ignore file* recipe

Creating and including local settings

You will need to have at least two different instances of your project: the development environment where you create new features and the public website environment in a hosted server. Additionally, there might be different development environments for other developers. Also, you might have a staging environment to test the project in a public-website-like situation.

Getting ready

Most of the settings for different environments will be shared and saved in version control. However, there will be some settings that are specific to the environment of the project instance, for example, database or e-mail settings. We will put them in the `local_settings.py` file.

How to do it...

Perform the following steps:

1. At the end of `settings.py`, add a version of `local_settings.py` that claims to be in the same directory, as follows:

```
#settings.py
# ... put this at the end of the file ...
try:
    execfile(os.path.join(os.path.dirname(__file__),
                          "local_settings.py"))
except IOError:
    pass
```

2. Create `local_settings.py` and put your environment-specific settings there, as follows:

```
#local_settings.py
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myproject",
        "USER": "root",
```

```
        "PASSWORD": "root",
    }
}
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
INSTALLED_APPS += (
    "debug_toolbar",
)
```

How it works...

As you can see, the local settings are not normally imported, but rather they are included and executed in the `settings.py` file itself. This allows you not only to create or overwrite the existing settings, but also to adjust the tuples or lists from the `settings.py` file; for example, we add `debug_toolbar` to `INSTALLED_APPS` here to be able to debug the SQL queries, template context variables, and so on.

See also

- ▶ The *Creating a project file structure* recipe
- ▶ The *Toggling Debug Toolbar* recipe in *Chapter 10, Bells and Whistles*

Setting UTF-8 as the default encoding for MySQL configuration

MySQL is the most popular open source database. In this recipe, I will tell you how to set UTF-8 as the default encoding for it. Note that if you don't set this encoding in the database configuration, you might get into a situation where LATIN1 is used by default with your UTF-8-encoded data. This will lead to database errors whenever symbols such as "€" are used. Also, this recipe will save you from the difficulties of converting database data from LATIN1 to UTF-8, especially when you have some tables encoded in LATIN1 and others in UTF-8.

Getting ready

Make sure that the MySQL database management system and the MySQLdb Python module are installed and you are using the MySQL engine in your project's settings.

How to do it...

Open the MySQL configuration file (`/etc/mysql/my.cnf`) in your favorite editor, and ensure that the following settings are set in the sections `[client]`, `[mysql]`, and `[mysqld]`:

```
#/etc/mysql/my.cnf
[client]
default-character-set = utf8

[mysql]
default-character-set = utf8

[mysqld]
collation-server = utf8_unicode_ci
init-connect = 'SET NAMES utf8'
character-set-server = utf8
```

If any of the sections don't exist, create them in the file. Then, restart MySQL in your command-line tool, as follows:

```
/etc/init.d/mysql restart
```

How it works...

Now, whenever you create a new MySQL database, the databases and all their tables will be set in the UTF-8 encoding by default.

Don't forget to set this in all computers where your project is developed or published.

Setting the Subversion ignore property

If you are using Subversion for version control, you will need to keep most of the projects in the repository, but some files and directories should stay only locally and not be tracked.

Getting ready

Make sure that your Django project is under the Subversion version control.

How to do it...

Open your command-line tool and set your default editor as `nano`, `vi`, `vim`, or any other that you prefer, as follows:

```
$ export EDITOR=nano
```



If you don't have a preference, I recommend using `nano`, which is very intuitive and a simple text editor for the terminal.

Then, go to your project directory and type the following command:

```
$ svn propedit svn:ignore .
```

This will open a temporary file in the editor, where you need to put the following file and directory patterns for Subversion to ignore:

```
*.pyc
local_settings.py
static
media
tmp
```

Save the file and exit the editor. For every other Python package in your project, you will need to ignore the `*.pyc` files. Just go to a directory and type the following command:

```
$ svn propedit svn:ignore .
```

Then, put this in the temporary file, save it, and close the editor:

```
*.pyc
```

How it works...

In Subversion, you need to define ignore properties for each directory of your project. Mainly, we don't want to track the Python-compiled files, for instance, `*.pyc`. We also want to ignore `local_settings.py` that is specific for each developer, `static` that replicates collected static files from different apps, `media` that contains uploaded files and will be changing together with the database, and `tmp` that is temporarily used for file uploads.

See also

- ▶ The *Creating and including local settings* recipe about including `local_settings.py` into your `settings.py` file
- ▶ The *Creating the Git ignore file* recipe

Creating the Git ignore file

If you are using Git—the most popular distributed version control system—ignoring some files and folders from version control is much easier than with Subversion.

Getting ready

Make sure that your Django project is under Git version control.

How to do it...

Using your favorite text editor, create a `.gitignore` file at the root of your Django project and put these files and directories there:

```
#.gitignore
*.pyc
/myproject/local_settings.py
/myproject/static/
/myproject/tmp/
/myproject/media/
```

How it works...

The `.gitignore` file specifies the paths that should intentionally be untracked by the Git version control system. The `.gitignore` file we created in this recipe will ignore the Python-compiled files, local settings, collected static files, temporary directory for uploads, and media directory with the uploaded files.

See also

- ▶ The *Setting the Subversion ignore property* recipe

Deleting Python-compiled files

When you run your project for the first time, Python compiles all your `*.py` code to bytecode-compiled files, `*.pyc`, which are used later for execution.

Normally, when you change the `*.py` files, `*.pyc` are recompiled, but sometimes when switching branches or moving directories, you need to clean up the compiled files manually.

Getting ready

Use your favorite editor and edit, or create, a `.bash_profile` file in your home directory.

How to do it...

Add this alias at the end of `.bash_profile`:

```
# ~/.bash_profile
alias delpyc="find . -name \"*.pyc\" -delete"
```

Now, to clean up the Python-compiled files, go to your project directory and type this command in the command line:

```
$ delpyc
```

How it works...

At first, we create a Unix alias that searches for the `*.pyc` files and deletes them in the current directory and its children. The `.bash_profile` file is executed when you start a new session in the command-line tool.

See also

- ▶ The *Setting the Subversion ignore property* recipe
- ▶ The *Creating the Git ignore file* recipe

Importing an order in Python files

When you create Python modules, it is good practice to stay consistent with the structure within the files. This makes it easier to read code for other developers and for yourself. This recipe will show you how to structure your imports.

Getting ready

Create a virtual environment and a Django project in it.

How to do it...

Use the following structure in any Python file you create. Just after the first line that defines UTF-8 as the default Python file encoding, put imports categorized into sections:

```
# -*- coding: UTF-8 -*-
# System libraries
import os
import re
from datetime import datetime

# Third-party libraries
import boto
from PIL import Image

# Django modules
from django.db import models
from django.conf import settings

# Django apps
from cms.models import Page

# Current-app modules
import app_settings
```

How it works...

We have five main categories for the imports, as follows:

- ▶ System libraries for packages within the default installation of Python
- ▶ Third-party libraries for additionally installed Python packages
- ▶ Django modules for different modules from the Django framework
- ▶ Django apps for third-party and local apps
- ▶ Current-app modules for relative imports from the current app

There's more...

When coding in Python and Django, use the official style guide for Python code, *PEP 8*. You can find it at <http://legacy.python.org/dev/peps/pep-0008/>.

See also

- ▶ *The Handling project dependencies with pip* recipe
- ▶ *The Including external dependencies in your project* recipe

Defining overwritable app settings

This recipe will show you how to define settings for your app that can then be overwritten in your project's `settings.py` or `local_settings.py` file. This is especially useful for reusable apps.

Getting ready

Create your Django app either manually or by using this command:

```
(myproject_env) $ django-admin.py startapp myapp1
```

How to do it...

If you have just one or two settings, you can use the following pattern in your `models.py` file. If the settings are extensive, and you want to have them better organized, create an `app_settings.py` file in the app and put the settings in the following way:

```
#models.py or app_settings.py
# -*- coding: UTF-8 -*-
from django.conf import settings
from django.utils.translation import ugettext_lazy as _

SETTING1 = getattr(settings, "MYAPP1_SETTING1", u"default value")
MEANING_OF_LIFE = getattr(settings, "MYAPP1_MEANING_OF_LIFE", 42)
STATUS_CHOICES = getattr(settings, "MYAPP1_STATUS_CHOICES", (
    ('draft', _("Draft")),
    ('published', _("Published")),
    ('not_listed', _("Not Listed")),
))
```

Then, you can use the app settings in `models.py` in the following way:

```
#models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

from app_settings import STATUS_CHOICES

class NewsArticle(models.Model):
    # ...
    status = models.CharField(_("Status"),
                              max_length=20, choices=STATUS_CHOICES
    )
```

If you want to overwrite the `STATUS_CHOICES` setting for just one project, you simply open `settings.py` and add this:

```
#settings.py
# ...
from django.utils.translation import ugettext_lazy as _
MYAPP1_STATUS_CHOICES = (
    ("imported", _("Imported")),
    ("draft", _("Draft")),
    ("published", _("Published")),
    ("not_listed", _("Not Listed")),
    ("expired", _("Expired")),
)
```

How it works...

The Python function, `getattr(object, attribute_name[, default_value])`, tries to get the `attribute_name` attribute from `object`, and returns `default_value` if it is not found. In this case, different settings are tried to be taken from the Django project settings module, and if they are not found, the default values are assigned.

2

Database Structure

In this chapter, we will cover the following topics:

- ▶ Using model mixins
- ▶ Creating a model mixin with URL-related methods
- ▶ Creating a model mixin to handle creation and modification dates
- ▶ Creating a model mixin to take care of meta tags
- ▶ Creating a model mixin to handle generic relations
- ▶ Handling multilingual fields
- ▶ Using South migrations
- ▶ Changing a foreign key to the many-to-many field with South

Introduction

When you start a new app, the first thing to do is create the models that represent your database structure. We are assuming that you have created Django apps before, or at least, you have read and understood the official Django tutorial. In this chapter, I will show you some interesting techniques that make your database structure consistent throughout different apps in your project. Then, I will show you how to create custom model fields to handle internationalization of your data in the database. At the end of the chapter, I will show you how to use migrations to change your database structure in the process of development.

Using model mixins

In object-oriented languages such as Python, a mixin class can be viewed as an interface with implemented features. When a model extends a mixin, it implements the interface and includes all its fields, properties, and methods. Mixins in Django models can be used when you want to reuse generic functionalities in different models again and again.

Getting ready

To start with, you will need to create some reusable mixins. Some typical examples of mixins will be shown later in this chapter. One good place to keep your model mixins is the `utils` module.



If you create a reusable app that you will share with others, keep model mixins within the app, for example, in the `base.py` file.

How to do it...

Open the `models.py` file of any Django app where you want to use mixins and type the following code:

```
#demo_app/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from utils.models import UrlMixin
from utils.models import CreationModificationMixin
from utils.models import MetaTagsMixin

class Idea(UrlMixin, CreationModificationMixin, MetaTagsMixin):
    title = models.CharField(_("Title"), max_length=200)
    content = models.TextField(_("Content"))

    class Meta:
        verbose_name = _("Idea")
        verbose_name_plural = _("Ideas")

    def __unicode__(self):
        return self.title
```

How it works...

Django model inheritance supports three types of inheritance: abstract base classes, multitable inheritance, and proxy models. Model mixins are abstract model classes with specified fields, properties, and methods. When you create a model such as `Idea`, as shown in the preceding example, it inherits all the features from `UrlMixin`, `CreationModificationMixin`, and `MetaTagsMixin`. All the fields of abstract classes are saved in the same database table as the fields of the extending model.

There's more...

To learn more about different types of model inheritance, refer to the official Django documentation available at <https://docs.djangoproject.com/en/1.6/topics/db/models/#model-inheritance>.

See also

- ▶ The *Creating a model mixin with URL-related methods* recipe
- ▶ The *Creating a model mixin to handle creation and modification dates* recipe
- ▶ The *Creating a model mixin to take care of meta tags* recipe

Creating a model mixin with URL-related methods

For each model that has its own page, it is good practice to define the `get_absolute_url()` method. This method can be used in templates and can also be used in the Django admin site to preview the saved object. However, `get_absolute_url()` is ambiguous because it really returns the URL path instead of the full URL. In this recipe, I will show you how to create a model mixin that allows you to define either the URL path or the full URL by default, generate the other one out of the box, and take care of the `get_absolute_url()` method being set.

Getting ready

If you haven't done it yet, create the `utils` package to save your mixins. Then, create the `models.py` file in the `utils` package (alternatively, if you create a reusable app, put the mixins in the `base.py` file in your app).

How to do it...

Execute the following steps one by one:

1. Add the following content to the `models.py` file of your `utils` package:

```
#utils/models.py
# -*- coding: UTF-8 -*-
import urlparse
from django.db import models
from django.contrib.sites.models import Site
from django.conf import settings

class UrlMixin(models.Model):
    """
    A replacement for get_absolute_url()
    Models extending this mixin should have either get_url or
    get_url_path implemented.
    """
    class Meta:
        abstract = True

    def get_url(self):
        if hasattr(self.get_url_path, "dont_recurse"):
            raise NotImplementedError
        try:
            path = self.get_url_path()
        except NotImplementedError:
            raise
        website_url = getattr(settings, "DEFAULT_WEBSITE_URL",
                              "http://127.0.0.1:8000")
        return website_url + path
    get_url.dont_recurse = True

    def get_url_path(self):
        if hasattr(self.get_url, "dont_recurse"):
            raise NotImplementedError
        try:
            url = self.get_url()
        except NotImplementedError:
            raise
        bits = urlparse.urlparse(url)
        return urlparse.urlunparse((" ", " ") + bits[2:])
    get_url_path.dont_recurse = True
```

```
def get_absolute_url(self):
    return self.get_url_path()
```

2. To use the mixin in your app, import it from the `utils` package, inherit the mixin in your model class, and define the `get_url_path()` method as follows:

```
#demo_app/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.core.urlresolvers import reverse

from utils.models import UrlMixin

class Idea(UrlMixin):
    title = models.CharField(_("Title"), max_length=200)

    # ...

    get_url_path(self):
        return reverse("idea_details", kwargs={
            "idea_id": str(self.pk),
        })
```

3. If you check this code in a staging or production environment or if you run a local server with a different IP or port than the defaults, set `DEFAULT_WEBSITE_URL` in your local settings (without the trailing slash), as follows:

```
#settings.py
# ...
DEFAULT_WEBSITE_URL = "http://www.example.com"
```

How it works...

The `UrlMixin` class is an abstract model that has three methods: `get_url()`, `get_url_path()`, and `get_absolute_url()`. The `get_url()` or `get_url_path()` methods are expected to be overwritten in the extended model class, for example, `Idea`. You can define `get_url()`, which is the full URL to the object, and then `get_url_path()` will strip it to the path. You can also define `get_url_path()`, which is the absolute path to the object, and then `get_url()` will add the website URL to the beginning of the path. The `get_absolute_url()` method will mimic the `get_url_path()` method.



The general rule of thumb is always to overwrite the `get_url_path()` method.



In the templates, use `{{ idea.title }}` `` when you need a link to an object in the same website. Use `{{ idea.title }}` `` for the links in e-mails, RSS feeds, or APIs.

See also

- ▶ The *Using model mixins* recipe
- ▶ The *Creating a model mixin to handle creation and modification dates* recipe
- ▶ The *Creating a model mixin to take care of meta tags* recipe
- ▶ The *Creating a model mixin to handle generic relations* recipe

Creating a model mixin to handle creation and modification dates

It is common behavior to have timestamps in your models for the creation and modification of your model instances. In this recipe, I will show you how to create a simple model mixin that saves creation and modification dates and times for your model. Using such a mixin will ensure that all the models use the same field names for the timestamps and have the same behavior.

Getting ready

If you haven't done this yet, create the `utils` package to save your mixins. Then, create the `models.py` file in the `utils` package.

How to do it...

Open the `models.py` file of your `utils` package and put the following content there:

```
#utils/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.utils.timezone import now as timezone_now

class CreationModificationDateMixin(models.Model):
    """
    Abstract base class with a creation and modification date and time
    """

    created = models.DateTimeField(
        _("creation date and time"),
```

```

        editable=False,
    )

    modified = models.DateTimeField(
        _("modification date and time"),
        null=True,
        editable=False,
    )

    def save(self, *args, **kwargs):
        if not self.pk:
            self.created = timezone_now()
        else:
            # To ensure that we have a creation data always,
            # we add this one
            if not self.created:
                self.created = timezone_now()

            self.modified = timezone_now()

        super(CreationModificationDateMixin, self).
            save(*args, **kwargs)
    save.alters_data = True

    class Meta:
        abstract = True

```

How it works...

The `CreationModificationDateMixin` class is an abstract model, which means that extending model classes will create all the fields in the same database table, that is, there will be no one-to-one relationships that make the table more difficult to handle. This mixin has two date-time fields and a `save()` method that will be called when saving the extended model. The `save()` method checks whether the model has no primary key, which is the case of a new not-yet-saved instance. In this case, it sets the creation date to the current date and time. Otherwise, if the primary key exists, the modification date is set to the current date and time.

Alternatively, instead of the `save()` method, you can use the `auto_now_add` and `auto_now` attributes for the `created` and `modified` fields, which will add creation and modification timestamps automatically.

See also

- ▶ The *Using model mixins* recipe
- ▶ The *Creating a model mixin to take care of meta tags* recipe
- ▶ The *Creating a model mixin to handle generic relations* recipe

Creating a model mixin to take care of meta tags

If you want to optimize your site for search engines, you need to set not only the semantic markup for each page, but also appropriate meta tags. For maximum flexibility, you need to have a way to define specific meta tags for each object that has its own page in your website. In this recipe, we will show you how to create a model mixin for the fields and methods related to meta tags.

Getting ready

As in the previous recipes, make sure that you have the `utils` package for your mixins. Open the `models.py` file from this package in your favorite editor.

How to do it...

Put the following content in the `models.py` file:

```
#utils/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.template.defaultfilters import escape
from django.utils.safestring import mark_safe

class MetaTagsMixin(models.Model):
    """
    Abstract base class for meta tags in the <head> section
    """
    meta_keywords = models.CharField(
        _('Keywords'),
        max_length=255,
        blank=True,
        help_text=_("Separate keywords by comma."),
    )
```

```
meta_description = models.CharField(
    _('Description'),
    max_length=255,
    blank=True,
)
meta_author = models.CharField(
    _('Author'),
    max_length=255,
    blank=True,
)
meta_copyright = models.CharField(
    _('Copyright'),
    max_length=255,
    blank=True,
)

class Meta:
    abstract = True

def get_meta_keywords(self):
    tag = u""
    if self.meta_keywords:
        tag = u'<meta name="keywords" content="%s" />\n' % \
            escape(self.meta_keywords)
    return mark_safe(tag)

def get_meta_description(self):
    tag = u""
    if self.meta_description:
        tag = u'<meta name="description" content="%s" />\n' % \
            escape(self.meta_description)
    return mark_safe(tag)

def get_meta_author(self):
    tag = u""
    if self.meta_author:
        tag = u'<meta name="author" content="%s" />\n' % \
            escape(self.meta_author)
    return mark_safe(tag)

def get_meta_copyright(self):
    tag = u""
    if self.meta_copyright:
        tag = u'<meta name="copyright" content="%s" />\n' % \
            escape(self.meta_copyright)
    return mark_safe(tag)
```

```
def get_meta_tags(self):
    return mark_safe(u"".join((
        self.get_meta_keywords(),
        self.get_meta_description(),
        self.get_meta_author(),
        self.get_meta_copyright(),
    )))
```

How it works...

This mixin adds four fields to the model extending from it: `meta_keywords`, `meta_description`, `meta_author`, and `meta_copyright`. The methods to render the meta tags in HTML are also added.

If you use this mixin in a model such as `Idea`, which is shown in the first recipe of this chapter, then you can put the following in the `HEAD` section of your detail page template for the purpose of rendering all meta tags:

```
{{ idea.get_meta_tags }}
```

You can also render a specific meta tag using the following line:

```
{{ idea.get_meta_description }}
```

As you might have noticed from the code snippet, the rendered meta tags are marked as `safe`, that is, they are not escaped and we don't need to use the `safe` template filter. Only the values coming from the database are escaped to guarantee that the final HTML is well formed.

See also

- ▶ The *Using model mixins* recipe
- ▶ The *Creating a model mixin to handle creation and modification dates* recipe
- ▶ The *Creating a model mixin to handle generic relations* recipe

Creating a model mixin to handle generic relations

Besides normal database relationships such as a foreign-key relationship or many-to-many relationship, Django has a mechanism to relate a model to an instance of any other model. This concept is called generic relations. For each generic relation, there is a content type of the related model saved as well as the ID of the instance of this model.

In this recipe, we will show you how to generalize the creation of generic relations into model mixins.

Getting ready

For this recipe to work, you need to have the `contenttypes` app installed. It should be in the `INSTALLED_APPS` directory by default:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "django.contrib.contenttypes",
)
```

Again, make sure that you have the `utils` package for your model mixins created.

How to do it...

Open the `models.py` file in the `utils` package in a text editor and put the following content there:

```
#utils/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes import generic
from django.core.exceptions import FieldError

def object_relation_mixin_factory(
    prefix=None,
    prefix_verbose=None,
```



```
        add_related_name=False,
        limit_content_type_choices_to={},
        limit_object_choices_to={},
        is_required=False,
    ):
        """
        returns a mixin class for generic foreign keys using
        "Content type - object Id" with dynamic field names.
        This function is just a class generator

        Parameters:
        prefix : a prefix, which is added in front of the fields
        prefix_verbose : a verbose name of the prefix, used to
                        generate a title for the field column
                        of the content object in the Admin.
        add_related_name : a boolean value indicating, that a
                        related name for the generated content
                        type foreign key should be added. This
                        value should be true, if you use more
                        than one ObjectRelationMixin in your model.
```

The model fields are created like this:

```
<<prefix>>_content_type : Field name for the "content type"
<<prefix>>_object_id : Field name for the "object Id"
<<prefix>>_content_object : Field name for the "content object"

"""
if prefix:
    p = "%s_" % prefix
else:
    p = ""

content_type_field = "%scontent_type" % p
object_id_field = "%sobject_id" % p
content_object_field = "%scontent_object" % p

class TheClass(models.Model):
    class Meta:
        abstract = True

    if add_related_name:
        if not prefix:
```

```

        raise FieldError("if add_related_name is set to True,"
                          "a prefix must be given")
        related_name = prefix
    else:
        related_name = None

    content_type = models.ForeignKey(
        ContentType,
        verbose_name=(prefix_verbose and _("%s's type (model)") % \
            prefix_verbose or _("Related object's type (model)")),
        related_name=related_name,
        blank=not is_required,
        null=not is_required,
        help_text=_("Please select the type (model) for the relation,"
                    "you want to build."),
        limit_choices_to=limit_content_type_choices_to,
    )

    object_id = models.CharField(
        (prefix_verbose or _("Related object")),
        blank=not is_required,
        null=False,
        help_text=_("Please enter the ID of the related object."),
        max_length=255,
        default="", # for south migrations
    )

    object_id.limit_choices_to = limit_object_choices_to
    # can be retrieved by
    #MyModel._meta.get_field("object_id").limit_choices_to

    content_object = generic.GenericForeignKey(
        ct_field=content_type_field,
        fk_field=object_id_field,
    )

    TheClass.add_to_class(content_type_field, content_type)
    TheClass.add_to_class(object_id_field, object_id)
    TheClass.add_to_class(content_object_field, content_object)

    return TheClass

```

The following is an example of how you can use two generic relationships in your app (copy this code in `demo_app/models.py`):

```
#demo_app/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from utils.models import object_relation_mixin_factory

FavoriteObjectMixin = object_relation_mixin_factory(
    is_required=True,
)

OwnerMixin = object_relation_mixin_factory(
    prefix="owner",
    prefix_verbose=_("Owner"),
    add_related_name=True,
    limit_content_type_choices_to={
        'model__in': ('user', 'institution')
    },
    is_required=True,
)

class Like(FavoriteObjectMixin, OwnerMixin):
    class Meta:
        verbose_name = _("Like")
        verbose_name_plural = _("Likes")

    def __unicode__(self):
        return _("%(owner)s likes %(obj)s") % {
            'owner': self.owner_content_object,
            'obj': self.content_object,
        }
```

How it works...

As you can see, this snippet is more complex than the previous ones. The `object_relation_mixin_factory` mixin is not a mixin itself; it is a function that generates a model mixin, that is, an abstract model class to extend from. The dynamically created mixin adds the `content_type` and `object_id` fields and the `content_object` generic foreign key that points to the related instance.

Why couldn't we just define a simple model mixin with these three attributes? A dynamically generated abstract class allows us to have prefixes for each field name, so we can have more than one generic relation in the same model. For example, the `Like` model, shown previously, will have the `content_type`, `object_id`, and `content_object` fields for the favorite object, and it'll also have `owner_content_type`, `owner_object_id`, and `owner_content_object` for the one (user or institution) who liked the object.

The `object_relation_mixin_factory` function adds a possibility to limit the content type choices by the `limit_content_type_choices_to` parameter. The preceding example limits the choices for `owner_content_type` only to the content types of the `User` and `Institution` models. Also, there is the `limit_object_choices_to` parameter, which can be used by a custom form validation to limit generic relations only to specific objects, for example, the objects with the `published` status.

See also

- ▶ The *Creating a model mixin with URL-related methods* recipe
- ▶ The *Creating a model mixin to handle creation and modification dates* recipe
- ▶ The *Creating a model mixin to take care of meta tags* recipe

Handling multilingual fields

Django uses the internationalization mechanism to translate verbose strings in the code and templates. However, it's up to the developer to decide how to implement multilingual content in the models. There are several third-party modules that handle translatable model fields, but I prefer the simple solution that will be introduced to you in this recipe.

The advantages of the approach you will learn here are as follows:

- ▶ It is straightforward to define multilingual fields in the database
- ▶ It is simple to use multilingual fields in database queries
- ▶ You can use contributed administration to edit models with multilingual fields without additional modifications
- ▶ If you need it, you can easily show all translations of an object in the same template
- ▶ You can use South database migrations to add or remove languages

Getting ready

Do you have the `utils` package created? You will need a new file, `fields.py`, for the custom model fields there.

How to do it...

Execute the following steps to define the multilingual character field and multilingual text field:

1. Open the `fields.py` file and create the multilingual character field as follows:

```
#utils/fields.py
# -*- coding: UTF-8 -*-
from django.conf import settings
from django.db import models
from django.utils.translation import get_language
from django.utils.translation import string_concat
from django.utils.encoding import force_unicode

class MultilingualCharField(models.CharField):
    def __init__(self, verbose_name=None, **kwargs):

        self._blank = kwargs.get('blank', False)
        self._editable = kwargs.get('editable', True)

        # inits for the needed dummy field (see below)
        kwargs['editable'] = False
        kwargs['null'] = True
        kwargs['blank'] = self._blank
        super(MultilingualCharField, self).__init__(verbose_name,
            **kwargs)

    def contribute_to_class(self, cls, name, virtual_only=False):
        # generate language specific fields dynamically
        if not cls._meta.abstract:
            for lang_code, lang_name in settings.LANGUAGES:
                if lang_code == settings.LANGUAGE_CODE:
                    _blank = self._blank
                else:
                    _blank = True

                try:
                    # the field shouldn't be already added
                    # (for south)
                    cls._meta.get_field("%s_%s" % \
                        (name, lang_code))
                except models.FieldDoesNotExist:
                    pass
                else:
                    continue
```

```

        localized_field = models.CharField(
            string_concat(self.verbose_name or name,
                u"(%s)" % lang_code),
            name=self.name,
            primary_key=self.primary_key,
            max_length=self.max_length,
            unique=self.unique,
            blank=_blank,
            null=False,
            # null value should be avoided for
            # text-based fields
            db_index=self.db_index,
            rel=self.rel,
            default=self.default or "",
            editable=self._editable,
            serialize=self.serialize,
            choices=self.choices,
            help_text=self.help_text,
            db_column=None,
            db_tablespace=self.db_tablespace
        )
        localized_field.contribute_to_class(
            cls,
            "%s_%s" % (name, lang_code),
        )

    super(MultilingualCharField, self).\
        contribute_to_class(cls, name, virtual_only=True)

    def translated_value(self):
        language = get_language()
        val = self.__dict__["%s_%s" % (name, language)]
        if not val:
            val = self.__dict__["%s_%s" % \
                (name, settings.LANGUAGE_CODE)]
        return val

    setattr(cls, name, property(translated_value))

```

2. In the same file, add an analogous multilingual text field. The differing parts are highlighted in the following code:

```
class MultilingualTextField(models.TextField):
    def __init__(self, verbose_name=None, **kwargs):
        self._blank = kwargs.get('blank', False)
        self._editable = kwargs.get('editable', True)

        # inits for the needed dummy field (see below)
        kwargs['editable'] = False
        kwargs['null'] = True
        kwargs['blank'] = self._blank
        super(MultilingualTextField, self).__init__(verbose_name,
            **kwargs)

    def contribute_to_class(self, cls, name, virtual_only=False):
        # generate language specific fields dynamically
        if not cls._meta.abstract:
            for lang_code, lang_name in settings.LANGUAGES:
                if lang_code == settings.LANGUAGE_CODE:
                    _blank = self._blank
                else:
                    _blank = True

                try:
                    # the field shouldn't be already added
                    # (for south)
                    cls._meta.get_field("%s_%s" % \
                        (name, lang_code))
                except models.FieldDoesNotExist:
                    pass
                else:
                    continue

                localized_field = models.TextField(
                    string_concat(self.verbose_name or name,
                        u"(%s)" % lang_code),
                    name=self.name,
                    primary_key=self.primary_key,
                    max_length=self.max_length,
                    unique=self.unique,
                    blank=_blank,
                    null=False,
                    # null value should be avoided for
                    # text-based fields
```

```

        db_index=self.db_index,
        rel=self.rel,
        default=self.default or "",
        editable=self._editable,
        serialize=self.serialize,
        choices=self.choices,
        help_text=self.help_text,
        db_column=None,
        db_tablespace=self.db_tablespace
    )
    localized_field.contribute_to_class(
        cls,
        "%s_%s" % (name, lang_code),
    )

    super(MultilingualTextField, self).\
        contribute_to_class(cls, name, virtual_only=True)

    def translated_value(self):
        language = get_language()
        val = self.__dict__["%s_%s" % (name, language)]
        if not val:
            val = self.__dict__["%s_%s" % \
                (name, settings.LANGUAGE_CODE)]
        return val

    setattr(cls, name, property(translated_value))

```

Now, we'll give you an example of how to use the multilingual fields in your app:

1. First, set multiple languages in your settings:

```

#myproject/settings.py
# -*- coding: UTF-8 -*-
# ...
LANGUAGE_CODE = 'en'

LANGUAGES = (
    ("en", u"English"),
    ("de", u"Deutsch"),
    ("fr", u"Français"),
    ("lt", u"Lietuvių kalba"),
)

```


2. Then, create multilingual fields for your model, as follows:

```
#demo_app/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Idea(models.Model):
    title = MultilingualCharField(
        _("Title"),
        max_length=200,
    )
    description = MultilingualTextField(
        _("Description"),
        blank=True,
    )

    class Meta:
        verbose_name = _("Idea")
        verbose_name_plural = _("Ideas")

    def __unicode__(self):
        return self.title
```

How it works...

The example of Idea will create a model like this:

```
class Idea(models.Model):
    title_en = models.CharField(
        _("Title (en)"),
        max_length=200,
    )
    title_de = models.CharField(
        _("Title (de)"),
        max_length=200,
        blank=True,
    )
    title_fr = models.CharField(
        _("Title (fr)"),
        max_length=200,
        blank=True,
    )
```

```

title_lt = models.CharField(
    _("Title (lt)"),
    max_length=200,
    blank=True,
)
description_en = models.TextField(
    _("Description (en)"),
    blank=True,
)
description_de = models.TextField(
    _("Description (de)"),
    blank=True,
)
description_fr = models.TextField(
    _("Description (fr)"),
    blank=True,
)
description_lt = models.TextField(
    _("Description (lt)"),
    blank=True,
)

```

In addition, there will be two properties, `title` and `description`, which will return the title and description in the currently active language.

The `MultilingualCharField` and `MultilingualTextField` fields are juggling model fields dynamically depending on your `LANGUAGES` setting. They overwrite the `contribute_to_class()` method that is used when the Django framework creates model classes. The multilingual fields set themselves as virtual fields so that they don't get a database field; they also add character or text fields for each language of the project dynamically. Also, the properties are created to return the translated value of the currently active language or the main language by default.

For example, you can have this in the template:

```

<h1>{{ idea.title }}</h1>
<div>{{ idea.description|urlize|linebreaks }}</div>

```

This will show the text in English, German, French, or Lithuanian, depending on the currently selected language, and will fall back to English if the translation doesn't exist.

Here is another example. If you want to have your query set ordered by translated titles in the view, you can define it like this:

```
qs = Idea.objects.order_by("title_%s" % request.LANGUAGE_CODE)
```

Using South migrations

It is wrong to think that once you create your database structure, it won't change in the future. As development happens iteratively, you can get updates on the business requirements in the development process and you will need to do database schema changes along the way. With the **django-south** app, you don't need to change database tables and fields manually because most of it is done automatically by using the command-line interface.

Getting ready

Install django-south into your project by putting it into `requirements.txt` or `externals` of your project. Put `south` into the installed apps in your project settings:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "south",
)
```

Activate your virtual environment in the command-line tool.

How to do it...

To create database migrations with the django-south app, have a look at the following steps:

1. When you create models in your new `demo_app` app, you need to create one initial migration that will create database tables for your app. This can be done using the following command:

```
(myproject_env)$ python manage.py schemamigration demo_app \
--initial
```

2. The first time you want to create all the tables of your project, run this command:

```
(myproject_env)$ python manage.py syncdb --migrate
```

It executes the usual database synchronization for all apps that have no database migrations, and in addition, it migrates all apps that have migrations set.

3. If you want to execute migrations for just a specific app, run this:

```
(myproject_env)$ python manage.py migrate demo_app
```

4. To execute migrations for all apps, run this command:

```
(myproject_env)$ python manage.py migrate
```

5. If you make some changes in the database schema, you have to create a migration for that schema. For example, if we add a new field `subtitle` to the `Idea` model, we can create the migration using this command:

```
(myproject_env)$ python manage.py schemamigration demo_app \
subtitle_added --auto
```

6. To create a data migration that modifies the data in the database table, we can use this command:

```
(myproject_env)$ python manage.py datamigration demo_app \
populate_subtitle
```

This creates a skeleton data migration, which you need to modify and add data manipulation to before applying.

7. To list all available applied and not-applied migrations, run this command:

```
(myproject_env)$ python manage.py migrate --list
```

The applied migrations will be listed with (*) prefixed.

How it works...

South migrations are instruction files for the South migration mechanism. The instruction files tell what database tables to create or remove, what fields to add or remove, and what data to insert, update, or delete.

There are two types of migrations in South. One of them is schema migration and the other is data migration. Schema migrations should be created when you add new models or add or remove fields. Data migrations should be used when you want to fill in the database with some values or when you want to massively delete values from the database. Data migrations should be created using a command in the command-line tool and then programmed in the migration file.

All migrations for each app are saved in the `migrations` directory. The first migration will be called `0001_initial.py`, and the other migrations can be called `0002_subtitle_added.py` and `0003_populate_subtitle.py`. Each migration gets a number prefix that is automatically incremented. For each migration that is executed, there is an entry saved in the `south_migrationhistory` database table.

It is possible to migrate back and forth by specifying the number of the migration to which to migrate to, as shown here:

```
(myproject_env)$ python manage.py migrate 0002
```

If you want to undo all the migrations for a specific app, you can do so by using the following command:

```
(myproject_env)$ python manage.py migrate zero
```



Do not commit your migrations to version control unless you have tested them and are sure that they will work well in other development and public-website environments.

There's more...

Django 1.7 comes with its own built-in migration system, which is not compatible with South. To learn more about the new migration operations, read the official Django documentation at <https://docs.djangoproject.com/en/1.7/ref/migration-operations/>.

See also

- ▶ The *Handling project dependencies with pip* and *Including external dependencies in your project* recipes in *Chapter 1, Getting Started with Django 1.6*
- ▶ The *Changing a foreign key to the many-to-many field with South* recipe

Changing a foreign key to the many-to-many field with South

This recipe is a practical example of how to change a many-to-one relation to a many-to-many relation preserving the already existing data. We will use both schema and data migrations for this situation.

Getting ready

Let's say you have the `Idea` model with a foreign key pointing to a `Category` model, as follows:

```
#demo_app/models.py
# -*- coding: UTF-8 -*-
from django.db import models

class Category(models.Model):
    title = models.CharField(_("Title"), max_length=200)
```

```
class Idea(models.Model):
    title = model.CharField(_("Title"), max_length=200)
    category = models.ForeignKey(Category, verbose_name=_("Category"),
                                null=True, blank=True)
```

The initial migration should be created and executed by using the following commands:

```
(myproject_env)$ python manage.py schemamigration demo_app --initial
(myproject_env)$ python manage.py syncdb --migrate
```

How to do it...

The following steps will teach you how to remove the foreign key relation while preserving the already existing data:

1. Add a new many-to-many field called `categories` as follows:

```
#demo_app/models.py
class Idea(models.Model):
    title = model.CharField(_("Title"), max_length=200)
    category = models.ForeignKey(Category,
                                verbose_name=_("Category"),
                                null=True,
                                blank=True,
                                )

    categories = models.ManyToManyField(Category,
                                       verbose_name=_("Categories"),
                                       blank=True,
                                       related_name="ideas",
                                       )
```

2. Create and run a schema migration to add the new field to the database, as follows:

```
(myproject_env)$ python manage.py schemamigration demo_app \
categories_added --auto
(myproject_env)$ python manage.py migrate demo_app
```

3. Create a data migration to copy categories from the foreign key to the many-to-many field, as follows:

```
(myproject_env)$ python manage.py datamigration demo_app \
copy_categories
```

4. Open the newly created migration file (`demo_app/migrations/0003_copy_categories.py`) and define the forward migration instructions, as follows:

```
#demo_app/migrations/0003_copy_categories.py
# -*- coding: utf-8 -*-
```

```
from south.utils import datetime_utils as datetime
from south.db import db
from south.v2 import DataMigration
from django.db import models

class Migration(DataMigration):
    def forwards(self, orm):
        for idea in orm.Idea.objects.all():
            if idea.category:
                idea.categories.add(idea.category)

    def backwards(self, orm):
        raise RuntimeError("Cannot reverse this migration.")
    #...
```

5. Run this data migration:

```
(myproject_env)$ python manage.py migrate demo_app
```

6. Delete the foreign key field `category` in the `models.py` file:

```
#demo_app/models.py
class Idea(models.Model):
    title = model.CharField(_("Title"), max_length=200)
    categories = models.ManyToManyField(Category,
        verbose_name=_("Categories"),
        blank=True,
        related_name="ideas",
    )
```

7. Create and run a schema migration to delete the `categories` field from the database, as follows:

```
(myproject_env)$ python manage.py schemamigration demo_app \
delete_category --auto
(myproject_env)$ python manage.py migrate demo_app
```

How it works...

At first, we add a new many-to-many field to the `Idea` model. Then, we copy existing relations from a foreign key relation to the many-to-many relation. Lastly, we remove the foreign key relation.

See also

- The *Using South migrations* recipe

3

Forms and Views

In this chapter, we will cover the following topics:

- ▶ Passing HttpRequest to the form
- ▶ Utilizing the save method of the form
- ▶ Uploading images
- ▶ Creating a form layout with django-crispy-forms
- ▶ Filtering object lists
- ▶ Managing paginated lists
- ▶ Composing class-based views
- ▶ Generating PDF documents

Introduction

When the database structure is defined in the models, we need some views to let users enter data or to show the data to the people. In this chapter, we will focus on the view managing forms, the list views, and views generating an alternative output than HTML. For the simplest examples, we will leave the creation of URL rules and templates up to you.

Passing HttpRequest to the form

The first argument of every Django view is the `HttpRequest` object usually named `request`. It contains metadata about the request, for example, current language code, current user, current cookies, or current session. By default, the forms used in the views accept the GET or POST parameters, files, initial data, and other parameters, but no `HttpRequest` object. In some cases, it is useful to pass `HttpRequest` additionally to the form, especially when you want to filter out choices of form fields using the request data or when you want to handle saving something like the current user or current IP in the form.

In this recipe, I will show you an example of a form where someone can choose a user and write a message to them. We will pass the `HttpRequest` object to the form to exclude the current user from recipient choices: we don't want anybody to write a message to themselves.

Getting ready

Let's create a new app called `email_messages`, and put it in to `INSTALLED_APPS` in the settings. This app will have no models, just forms and views.

How to do it...

1. Add a new file, `forms.py`, with the message form containing two fields: the recipient selection and message text. Also, this form will have an initialization method, which will accept the `request` object and then modify the `QuerySet` for the recipient's selection field:

```
#email_messages/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from django.utils.translation import ugettext_lazy as _
from django.contrib.auth.models import User

class MessageForm(forms.Form):
    recipient = forms.ModelChoiceField(
        label=_("Recipient"),
        queryset=User.objects.all(),
        required=True,
    )
    message = forms.CharField(
        label=_("Message"),
        widget=forms.Textarea,
        required=True,
    )
```

```
def __init__(self, request, *args, **kwargs):
    super(MessageForm, self).__init__(*args, **kwargs)
    self.request = request
    self.fields['recipient'].queryset = \
        self.fields['recipient'].queryset.\
            exclude(pk=request.user.pk)
```

2. Then, create `views.py` with the `message_to_user` view to handle the form. As you can see, the `request` object is passed as the first parameter to the form:

```
#email_messages/views.py
# -*- coding: UTF-8 -*-
from django.contrib.auth.decorators import login_required
from django.shortcuts import render, redirect

from forms import MessageForm

@login_required
def message_to_user(request):
    if request.method == "POST":
        form = MessageForm(request, data=request.POST)
        if form.is_valid():
            # do something with the form
            return redirect("message_to_user_done")
    else:
        form = MessageForm(request)

    return render(request, "email_messages/message_to_user.html",
        {'form': form})
```

How it works...

In the initialization method, we have the `self` variable that represents the instance of the form itself, then we have the newly added `request` variable, and then we have the rest of the positional arguments (`*args`) and named arguments (`**kwargs`). We call the super initialization method passing all the positional and named arguments to it so that the form is properly initiated. And then, we assign the `request` variable to a new `request` attribute of the form for later access in other methods of the form. Then, we modify the `queryset` attribute of the recipient's selection field excluding the current user from the request.

In the view, we are passing the `HttpRequest` object as the first argument in both situations: when the form is posted as well as when the form is loaded for the first time.

See also

- ▶ The *Utilizing the save method of the form* recipe

Utilizing the save method of the form

To make your views clean and simple, it is a good practice to move the handling of the form data to the form itself, whenever possible and makes sense. The common practice is to have a `save` method that will save the data, perform search, or do some other smart actions. We will extend the form defined in the previous recipe with the `save` method, which will send an e-mail to the selected recipient.

Getting ready

We will start with the example defined in the *Passing HttpRequest to the form* recipe.

How to do it...

Execute these two steps:

1. Import the function to send an e-mail in the forms of your app. Then add the `save` method that tries to send an e-mail to the selected recipient and fails silently with an error:

```
#email_messages/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from django.utils.translation import ugettext, ugettext_lazy as _
from django.core.mail import send_mail
from django.contrib.auth.models import User

class MessageForm(forms.Form):
    recipient = forms.ModelChoiceField(
        label=_("Recipient"),
        queryset=User.objects.all(),
        required=True,
    )
    message = forms.CharField(
        label=_("Message"),
        widget=forms.Textarea,
        required=True,
    )
```

```

def __init__(self, request, *args, **kwargs):
    super(MessageForm, self).__init__(*args, **kwargs)
    self.request = request
    self.fields['recipient'].queryset = \
        self.fields['recipient'].queryset.\
            exclude(pk=request.user.pk)

def save(self):
    cleaned_data = self.cleaned_data
    send_mail(
        subject=ugettext("A message from %s") % \
            self.request.user,
        message=cleaned_data['message'],
        from_email=self.request.user.email,
        recipient_list=[cleaned_data['recipient'].email],
        fail_silently=True,
    )

```

2. Finally, call the save method from the form in the view, if the posted data is valid:

```

#email_messages/views.py
# -*- coding: UTF-8 -*-
from django.contrib.auth.decorators import login_required
from django.shortcuts import render, redirect

from forms import MessageForm

@login_required
def message_to_user(request):
    if request.method == "POST":
        form = MessageForm(request, data=request.POST)
        if form.is_valid():
            form.save()
            return redirect("message_to_user_done")
    else:
        form = MessageForm(request)

    return render(request, "email_messages/message_to_user.html",
        {'form': form})

```

How it works...

Let's look at the form at first. The `save` method uses the cleaned data from the form to read the recipient's e-mail address and the e-mail message. The sender of the e-mail is the current user from the request. If the e-mail cannot be sent because of an incorrect mail server configuration or another reason, it will fail silently, that is, no error will be raised.

Now, let's look at the view. When the posted form is valid, the `save` method of the form will be called and then the user will be redirected to the success page.

See also

- ▶ *The Passing HttpRequest to the form recipe*

Uploading images

In this recipe, we will have a look at the easiest way to handle image uploads. You will see an example of an app where visitors can upload images with inspirational quotes.

Getting ready

First of all, let's create an app, `quotes`, and put it in to `INSTALLED_APPS` in the settings. Then, we will add an `InspirationalQuote` model with three fields: the author, the quote text, and the picture as follows:

```
#quotes/models.py
# -*- coding: UTF-8 -*-
import os
from django.db import models
from django.utils.timezone import now as timezone_now
from django.utils.translation import ugettext_lazy as _

def upload_to(instance, filename):
    now = timezone_now()
    filename_base, filename_ext = os.path.splitext(filename)
    return 'quotes/%s%s' % (
        now.strftime("%Y/%m/%Y%m%d%H%M%S"),
        filename_ext.lower(),
    )

class InspirationQuote(models.Model):
```

```

author = models.CharField(_("Author"), max_length=200)
quote = models.TextField(_("Quote"))
picture = models.ImageField(_("Picture"),
                             upload_to=upload_to,
                             blank=True,
                             null=True,
                             )

class Meta:
    verbose_name = _("Inspiration Quote")
    verbose_name_plural = _("Inspiration Quotes")

def __unicode__(self):
    return self.quote

```

In addition, we created a function, `upload_to`, which sets the path of the uploaded picture to be something like `quotes/2014/04/20140424140000.png`. As you can see, we use the date timestamp as the filename to ensure its uniqueness. We pass this function to the picture image field.

How to do it...

Create the `forms.py` file and put a simple model form there:

```

#quotes/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from models import InspirationQuote

class InspirationQuoteForm(forms.ModelForm):
    class Meta:
        model = InspirationQuote

```

In the `views.py` file, put a view that handles the form. Don't forget to pass the `FILES` dictionary-like object to the form. When the form is valid, trigger the `save` method as follows:

```

#quotes/views.py
# -*- coding: UTF-8 -*-
from django.shortcuts import redirect
from django.shortcuts import render
from forms import InspirationQuoteForm

def add_quote(request):
    if request.method == "POST":
        form = InspirationQuoteForm(
            data=request.POST,

```

```
        files=request.FILES,
    )
    if form.is_valid():
        quote = form.save()
        return redirect("add_quote_done")
    else:
        form = InspirationQuoteForm()
    return render(request, "quotes/change_quote.html", {'form': form})
```

Lastly, create a template for the view in `templates/quotes/change_quote.html`. It is very important to set the `enctype` attribute to `"multipart/form-data"` for the HTML form, otherwise the file upload won't work:

```
{% extends "base.html" %}
{% load i18n %}

{% block content %}
    <form method="post" action="" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">{% trans "Save" %}</button>
    </form>
{% endblock %}
```

How it works...

Django model forms are forms created from models. They provide all the fields from the model, so you don't need to define them again. In the preceding example, we created a model form for the `InspirationQuote` model. When we save the form, the form knows how to save each field to the database as well as how to upload the files and save them in the media directory.

There's more

As a bonus, I will show you an example of how to generate a thumbnail out of the uploaded image. Using this technique, you could also generate several other specific versions of the image such as the list version, the mobile version, and the desktop computer version.

We will add three methods to the `InspirationQuote` model (`quotes/models.py`). They are `save`, `create_thumbnail`, and `get_thumbnail_picture_url`. When the model is being saved, we trigger the creation of the thumbnail. When we need to show the thumbnail in a template, we can get its URL by using `{{ quote.get_thumbnail_picture_url }}` as follows:

```
#quotes/models.py
class InspirationQuote(models.Model):
    # ...
    def save(self, *args, **kwargs):
```

```

        super(InspirationQuote, self).save(*args, **kwargs)
        # generate thumbnail picture version
        self.create_thumbnail()

def create_thumbnail(self):
    from django.core.files.storage import default_storage as \
        storage
    if not self.picture:
        return ""
    file_path = self.picture.name
    filename_base, filename_ext = os.path.splitext(file_path)
    thumbnail_file_path = "%s_thumbnail.jpg" % filename_base
    if storage.exists(thumbnail_file_path):
        # if thumbnail version exists, return its url path
        return "exists"
    try:
        # resize the original image and
        # return URL path of the thumbnail version
        f = storage.open(file_path, 'r')
        image = Image.open(f)
        width, height = image.size
        thumbnail_size = 50, 50

        if width > height:
            delta = width - height
            left = int(delta/2)
            upper = 0
            right = height + left
            lower = height
        else:
            delta = height - width
            left = 0
            upper = int(delta/2)
            right = width
            lower = width + upper

        image = image.crop((left, upper, right, lower))
        image = image.resize(thumbnail_size, Image.ANTIALIAS)

        f_mob = storage.open(thumbnail_file_path, "w")
        image.save(f_mob, "JPEG")
        f_mob.close()
        return "success"

```



```
except:
    return "error"

def get_thumbnail_picture_url(self):
    from PIL import Image
    from django.core.files.storage import default_storage as \
        storage
    if not self.picture:
        return ""
    file_path = self.picture.name
    filename_base, filename_ext = os.path.splitext(file_path)
    thumbnail_file_path = "%s_thumbnail.jpg" % filename_base
    if storage.exists(thumbnail_file_path):
        # if thumbnail version exists, return its URL path
        return storage.url(thumbnail_file_path)
    # return original as a fallback
    return self.picture.url
```

In the preceding methods, we are using the file storage API instead of directly juggling the filesystem, because then we could exchange the default storage with Amazon S3 buckets or other storage services and the methods will still work.

How does the creating of the thumbnail work? If we had the original saved as `quotes/2014/04/20140424140000.png`, we are checking if the file `quotes/2014/04/20140424140000_thumbnail.jpg` doesn't exist and in that case, we are opening the original image, cropping it from the center, resizing it to 50 by 50 pixels, and saving to the storage.

The `get_thumbnail_picture_url` method checks whether the thumbnail version exists in the storage and returns its URL. If the thumbnail version does not exist, the URL of the original image is returned as a fallback.

See also

- *The Creating form layout with django-crispy-forms recipe*

Creating a form layout with django-crispy-forms

The Django app, **django-crispy-forms** allows you to build, customize, and reuse forms using one of the following CSS frameworks: **Uni-Form**, **Bootstrap**, or **Foundation**. The usage of django-crispy-forms is analogous to fieldsets in the Django contributed administration, but it is more advanced and customizable. You define form layout in Python code and you don't need to care much about how each field is presented in HTML. Still, if you need to add specific HTML attributes or specific wrapping, you can easily do that too.

In this recipe, we will show you an example of how to use django-crispy-forms with Bootstrap 3, which is the most popular frontend framework for developing responsive, mobile-first web projects.

Getting ready

To start with, execute these tasks one by one:

1. Download the Bootstrap front end framework from <http://getbootstrap.com/> and integrate the CSS and JavaScript into the templates. See more about that in the *Arranging the base.html template* recipe in *Chapter 4, Templates and JavaScript*.
2. Install django-crispy-forms in your virtual environment using the following command:
`(myproject_env)$ pip install django-crispy-forms`
3. Make sure that `crispy_forms` is added to `INSTALLED_APPS` and then set `bootstrap3` as the template pack used in this project:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "crispy_forms",
)
# ...
CRISPY_TEMPLATE_PACK = "bootstrap3"
```

4. Let's create a `bulletin_board` app to illustrate the usage of `django-crispy-forms` and put it into `INSTALLED_APPS` in the settings. We will have a `Bulletin` model there with these fields: type, title, description, contact person, phone, e-mail, and image:

```
#bulletin_board/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

TYPE_CHOICES = (
    ('searching', _("Searching")),
    ('offering', _("Offering")),
)

class Bulletin(models.Model):
    bulletin_type = models.CharField(_("Type"), max_length=20,
                                     choices=TYPE_CHOICES)

    title = models.CharField(_("Title"), max_length=255)
    description = models.TextField(_("Description"),
                                   max_length=300)

    contact_person = models.CharField(_("Contact person"),
                                      max_length=255)
    phone = models.CharField(_("Phone"), max_length=200,
                             blank=True)
    email = models.EmailField(_("Email"), blank=True)

    image = models.ImageField(_("Image"), max_length=255,
                              upload_to="bulletin_board/", blank=True)

    class Meta:
        verbose_name = _("Bulletin")
        verbose_name_plural = _("Bulletins")
        ordering = ("title",)

    def __unicode__(self):
        return self.title
```

How to do it...

Let's add a model form for the bulletin in the newly created app. We will attach a form helper to the form itself in the initialization method. The form helper will have the `layout` property, which will define the layout for the form, as follows:

```
#bulletin_board/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from django.utils.translation import ugettext_lazy as _, ugettext
from crispy_forms.helper import FormHelper
from crispy_forms import layout, bootstrap
from models import Bulletin

class BulletinForm(forms.ModelForm):
    class Meta:
        model = Bulletin
        fields = ['bulletin_type', 'title', 'description',
                  'contact_person', 'phone', 'email', 'image']

    def __init__(self, *args, **kwargs):
        super(BulletinForm, self).__init__(*args, **kwargs)

        self.helper = FormHelper()
        self.helper.form_action = ""
        self.helper.form_method = "POST"

        self.fields['bulletin_type'].widget = forms.RadioSelect()
        # delete empty choice for the type
        del self.fields['bulletin_type'].choices[0]

        self.helper.layout = layout.Layout(
            layout.Fieldset(
                _("Main data"),
                layout.Field("bulletin_type"),
                layout.Field("title", css_class="input-block-level"),
                layout.Field("description",
                             css_class="input-blocklevel", rows="3"),
            ),
            layout.Fieldset(
                _("Image"),
                layout.Field("image", css_class="input-block-level"),
                layout.HTML(u"""{% load i18n %}"),
            )
        )
```

```
        <p class="help-block">{% trans "Available formats
are JPG, GIF, and PNG. Minimal size is 800 x 800 px." %}</p>
        """),
        title=_("Image upload"),
        css_id="image_fieldset",
    ),
    layout.Fieldset(
        _("Contact"),
        layout.Field("contact_person",
            css_class="input-blocklevel"),
        layout.Div(
            bootstrap.PrependText("phone", ""<span
class="glyphicon glyphicon-earphone"></span>""),
            css_class="inputblock-level"),
            bootstrap.PrependText("email", "@",
            css_class="input-block-level",
            placeholder="contact@example.com"),
            css_id="contact_info",
        ),
    ),
    bootstrap.FormActions(
        layout.Submit('submit', _('Save')),
    )
)
```

To render the form in the template, we just need to load the `crispy_forms_tags` template-tag library and use the `{% crispy %}` template tag as follows:

```
{#templates/bulletin_board/change_form.html}
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block content %}
    {% crispy form %}
{% endblock %}
```

How it works...

The page with the bulletin form will look like this:

The screenshot shows a web browser window with the title 'My Website' and the URL '127.0.0.1:8000/bulletin-board/'. The page content is as follows:

My Website

Main data

Type*

☐ Searching

☐ Offering

Title*

Description*

Image

Image

No file chosen

Available formats are JPG, GIF, and PNG. Minimal size is 800 x 800 px.

Contact

Contact person*

Phone

Email

As you see, the fields are grouped by fieldsets. The first argument of the `Fieldset` object defines the legend, the other positional arguments define fields. You can also pass named arguments to define HTML attributes for the fieldset; for example, for the second fieldset, we are passing `title` and `css_id` to set the HTML attributes `title` and `id`.

Fields can also have additional attributes passed by named arguments, for example, for the description field, we are passing `css_class` and `rows` to set the HTML attributes `class` and `rows`.

Besides the normal fields, you can pass HTML snippets as this is done with the `help` block for the `image` field. You can also have prepended-text fields in the layout, for example, we added a phone icon to the `phone` field, and an @ sign for the `email` field. As you see from the example with contact fields, we can easily wrap fields into HTML `<div>` elements using `Div` objects. This is useful when specific JavaScript needs to be applied to some form fields.

The `action` attribute for the HTML form is defined by the `form_action` property of the form helper. The `method` attribute of the HTML form is defined by the `form_method` property of the form helper. Finally, there is a `Submit` object to render the submit button, which takes the name of the button as the first positional argument, and the value of the button as the second argument.

There's more...

For the basic usage, the given example is more than necessary. However, if you need specific markup for forms in your project, you can still overwrite and modify templates of the `django-crispy-forms` app, as there is no markup hardcoded in Python files, but rather all the generated markup is rendered through the templates. Just copy the templates from the `django-crispy-forms` app to your project's template directory and change them as you need.

See also

- ▶ *The Filtering object lists recipe*
- ▶ *The Managing paginated lists recipe*

Filtering object lists

In web development, besides views with forms, it is typical to have object-list views and detail views. List views can simply list objects ordered, for example, alphabetically or by creation date, but that is not very user friendly with huge amounts of data. For the best accessibility and convenience, you should be able to filter the content by all possible categories. In this recipe, I will show you the pattern used to filter list views by any number of categories.

Getting ready

For the filtering example, we will use the `Movie` model with relations to genres, directors, and actors to filter by. It will also be possible to filter by ratings, which is `PositiveIntegerField` with choices. Let's create the `movies` app, put it into `INSTALLED_APPS` in the settings (`movies/models.py`), and define the mentioned models in the new app:

```
#movies/models.py
# -*- coding: UTF-8 -*-
```

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

RATING_CHOICES = (
    (1, u"★"),
    (2, u"★★"),
    (3, u"★★★"),
    (4, u"★★★★"),
    (5, u"★★★★★"),
)

class Genre(models.Model):
    title = models.CharField(_("Title"), max_length=100)

    def __unicode__(self):
        return self.title

class Director(models.Model):
    first_name = models.CharField(_("First name"), max_length=40)
    last_name = models.CharField(_("Last name"), max_length=40)

    def __unicode__(self):
        return self.first_name + " " + self.last_name

class Actor(models.Model):
    first_name = models.CharField(_("First name"), max_length=40)
    last_name = models.CharField(_("Last name"), max_length=40)

    def __unicode__(self):
        return self.first_name + " " + self.last_name

class Movie(models.Model):
    title = models.CharField(_("Title"), max_length=255)
    genres = models.ManyToManyField(Genre, blank=True)
    directors = models.ManyToManyField(Director, blank=True)
    actors = models.ManyToManyField(Actor, blank=True)
    rating = models.PositiveIntegerField(choices=RATING_CHOICES)

    def __unicode__(self):
        return self.title
```


How to do it...

First of all, we create `MovieFilterForm` with all the possible categories to filter by:

```
#movies/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from django.utils.translation import ugettext_lazy as _

from models import Genre
from models import Director
from models import Actor
from models import RATING_CHOICES

class MovieFilterForm(forms.Form):
    genre = forms.ModelChoiceField(
        label=_("Genre"),
        required=False,
        queryset=Genre.objects.all(),
    )
    director = forms.ModelChoiceField(
        label=_("Director"),
        required=False,
        queryset=Director.objects.all(),
    )
    actor = forms.ModelChoiceField(
        label=_("Actor"),
        required=False,
        queryset=Actor.objects.all(),
    )
    rating = forms.ChoiceField(
        label=_("Rating"),
        required=False,
        choices=RATING_CHOICES,
    )
```

Then, we create a `movie_list` view that will use `MovieFilterForm` to validate the request query parameters and do the filtering by chosen categories. Note the `facets` dictionary, which is used here to list the categories and also the currently selected choices:

```
#movies/views.py
# -*- coding: UTF-8 -*-
from django.shortcuts import render
from models import Genre
from models import Director
from models import Actor
from models import Movie, RATING_CHOICES
from forms import MovieFilterForm

def movie_list(request):
    qs = Movie.objects.order_by('title')

    form = MovieFilterForm(data=request.REQUEST)

    facets = {
        'selected': {},
        'categories': {
            'genres': Genre.objects.all(),
            'directors': Director.objects.all(),
            'actors': Actor.objects.all(),
            'ratings': RATING_CHOICES,
        },
    }

    if form.is_valid():
        genre = form.cleaned_data['genre']
        if genre:
            facets['selected']['genre'] = genre
            qs = qs.filter(genres=genre).distinct()

        director = form.cleaned_data['director']
        if director:
            facets['selected']['director'] = director
            qs = qs.filter(directors=director).distinct()

        actor = form.cleaned_data['actor']
        if actor:
```

```
facets['selected']['actor'] = actor
qs = qs.filter(actors=actor).distinct()

rating = form.cleaned_data['rating']
if rating:
    facets['selected']['rating'] = \
        (int(rating), dict(RATING_CHOICES[int(rating)]))
    qs = qs.filter(rating=rating).distinct()

context = {
    'form': form,
    'facets': facets,
    'object_list': qs,
}
return render(request, "movies/movie_list.html", context)
```

Lastly, we create the template for the list view. We will use the `facets` dictionary here to list the categories and to know which category is currently selected. To generate URLs for the filters, we will use the `{% append_to_query %}` template tag, which will be described later in the *Creating a template tag to modify request query parameters* recipe in *Chapter 5, Custom Template Filters and Tags*. Copy the following code in the `templates/movies/movie_list.html` directory:

```
{% templates/movies/movie_list.html %}
{% extends "base_two_columns.html" %}
{% load i18n utility_tags %}

{% block sidebar %}
<div class="filters">
  <h6>{% trans "Filter by Genre" %}</h6>
  <div class="list-group">
    <a class="list-group-item{% if not facets.selected.genre %}
active{% endif %}" href="{% append_to_query genre="" page="" %}">{%
trans "All" %}</a>
    {% for cat in facets.categories.genres %}
      <a class="list-group-item{% if facets.selected.genre
== cat %} active{% endif %}" href="{% append_to_query genre=cat.pk
page="" %}">{{ cat }}</a>
    {% endfor %}
  </div>

  <h6>{% trans "Filter by Director" %}</h6>
  <div class="list-group">
```

```

        <a class="list-group-item{% if not facets.selected.director %}
active{% endif %}" href="{% append_to_query director="" page="" %}">{%
trans "All" %}</a>
        {% for cat in facets.categories.directors %}
            <a class="list-group-item{% if facets.selected.director
== cat %} active{% endif %}" href="{% append_to_query director=cat.pk
page="" %}">{{ cat }}</a>
            {% endfor %}
        </div>

        <h6>{% trans "Filter by Actor" %}</h6>
        <div class="list-group">
            <a class="list-group-item{% if not facets.selected.actor %}
active{% endif %}" href="{% append_to_query actor="" page="" %}">{%
trans "All" %}</a>
            {% for cat in facets.categories.actors %}
                <a class="list-group-item{% if facets.selected.actor
== cat %} active{% endif %}" href="{% append_to_query actor=cat.pk
page="" %}">{{ cat }}</a>
                {% endfor %}
            </div>

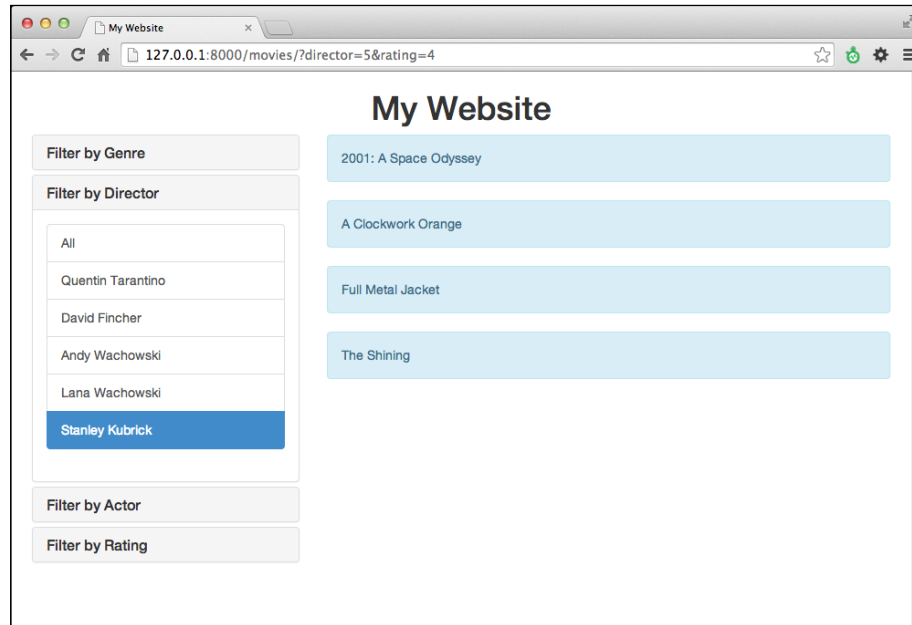
            <h6>{% trans "Filter by Rating" %}</h6>
            <div class="list-group">
                <a class="list-group-item{% if not facets.selected.rating %}
active{% endif %}" href="{% append_to_query rating="" page="" %}">{%
trans "All" %}</a>
                {% for r_val, r_display in facets.categories.ratings %}
                    <a class="list-group-item{% if facets.selected.rating.0
== r_val %} active{% endif %}" href="{% append_to_query rating=r_val
page="" %}">{{ r_display }}</a>
                    {% endfor %}
                </div>
            </div>
        {% endblock %}

        {% block content %}
        <div class="movie_list">
            {% for movie in object_list %}
                <div class="movie">
                    <h3>{{ movie.title }}</h3>
                </div>
            {% endfor %}
        </div>
        {% endblock %}

```

How it works...

If we use the Bootstrap 3 frontend framework, the end result will look like this in the browser with some filters applied:



So, we are using the `facets` dictionary that is passed to the template context, to know what filters we have and which filters are selected. To look deeper, the `facets` dictionary consists of two sections: the `categories` dictionary and the `selected` dictionary. The `categories` dictionary contains the `QuerySets` or choices of all filterable categories. The `selected` dictionary contains the currently selected values for each category.

In the view, we check if the query parameters are valid in the form and then we drill down the `QuerySet` of objects by the selected categories. Additionally, we set the selected values to the `facets` dictionary, which will be passed to the template.

In the template, for each categorization from the `facets` dictionary, we list all categories and mark the currently selected category as active.

It is as simple as that.

See also

- ▶ The *Managing paginated lists* recipe
- ▶ The *Composing class-based views* recipe
- ▶ The *Creating a template tag to modify request query parameters* recipe in Chapter 5, *Custom Template Filters and Tags*

Managing paginated lists

If you have dynamically changing lists of objects or when the amount of them can be greater than 30-50, you surely need pagination for the list. With pagination instead of the full `QuerySet`, you provide a fraction of the dataset limited to a specific amount per page and you also show the links to get to the other pages of the list. Django has classes to manage paginated data, and in this recipe, I will show you how to do that for the example from the previous recipe.

Getting ready

Let's start with the `movies` app and the forms as well as the views from the *Filtering object lists* recipe.

How to do it...

At first, import the necessary pagination classes from Django. We will add pagination management to the `movie_list` view just after filtering. Also, we will slightly modify the context dictionary by passing a page instead of the movie `QuerySet` as `object_list`:

```
#movies/views.py
# -*- coding: UTF-8 -*-
from django.shortcuts import render
from django.core.paginator import Paginator, EmptyPage, \
    PageNotAnInteger

from models import Movie
from forms import MovieFilterForm

def movie_list(request):
    qs = Movie.objects.order_by('title')
    # ... filtering goes here...

    paginator = Paginator(qs, 15)
```

```
page_number = request.GET.get('page')
try:
    page = paginator.page(page_number)
except PageNotAnInteger:
    # If page is not an integer, show first page.
    page = paginator.page(1)
except EmptyPage:
    # If page is out of range, show last existing page.
    page = paginator.page(paginator.num_pages)

context = {
    'form': form,
    'object_list': page,
}
return render(request, "movies/movie_list.html", context)
```

In the template, we will add pagination controls after the list of movies as follows:

```
{% templates/movies/movie_list.html %}
{% extends "base.html" %}
{% load i18n utility_tags %}

{% block sidebar %}
    {# ... filters go here... #}
{% endblock %}

{% block content %}
<div class="movie_list">
    {% for movie in object_list %}
        <div class="movie alert alert-info">
            <p>{{ movie.title }}</p>
        </div>
    {% endfor %}
</div>

{% if object_list.has_other_pages %}
    <ul class="pagination">
        {% if object_list.has_previous %}
            <li><a href="{% append_to_query page=object_list.previous_
page_number %}">&laquo;</a></li>
        {% else %}
            <li class="disabled"><span>&laquo;</span></li>
        {% endif %}
    </ul>
{% endif %}
```

```

{% for page_number in object_list.paginator.page_range %}
    {% if page_number == object_list.number %}
        <li class="active">
            <span>{{ page_number }} <span class="sr-
only">(current)</span></span>
        </li>
    {% else %}
        <li>
            <a href="{% append_to_query page=page_number
%}">{{ page_number }}</a>
        </li>
    {% endif %}
{% endfor %}
{% if object_list.has_next %}
    <li><a href="{% append_to_query page=object_list.next_
page_number %}">&raquo;</a></li>
{% else %}
    <li class="disabled"><span>&raquo;</span></li>
{% endif %}
</ul>
{% endif %}

{% endblock %}

```

How it works...

When you look at the results in the browser, you will see pagination controls like these, added after the list of movies:



How do we achieve that? When the `QuerySet` is filtered out, we create a `paginator` object passing the `QuerySet` and the maximal amount of items we want to show per page (which is 15 here). Then, we read the current page number from the query parameter, `page`. The next step is retrieving the current page object from the paginator. If the page number was not an integer, we get the first page. If the number exceeds the amount of possible pages, the last page is retrieved. The page object has methods and attributes necessary for the pagination widget shown in the preceding screenshot. Also, the page object acts like a `QuerySet`, so that we can iterate through it and get the items from the fraction of the page.

The snippet marked in the template creates a pagination widget with the markup for the Bootstrap 3 frontend framework. We show the pagination controls only if there are more pages than the current one. We have the links to the previous and next pages, and the list of all page numbers in the widget. The current page number is marked as active. To generate URLs for the links, we are using the template tag `{% append_to_query %}`, which will be described later in the *Creating a template tag to modify request query parameters* recipe in Chapter 5, *Custom Template Filters and Tags*.

See also

- ▶ The *Filtering object lists* recipe
- ▶ The *Composing class-based views* recipe
- ▶ The *Creating a template tag to modify request query parameters* recipe in Chapter 5, *Custom Template Filters and Tags*

Composing class-based views

Django views are callables that take requests and return responses. In addition to function-based views, Django provides an alternative way to define views as classes. This approach is useful when you want to create reusable modular views or when you want to combine views out of generic mixins. In this recipe, we will convert the previously shown function-based view, `movie_list`, into a class-based view, `MovieListView`.

Getting ready

Create the models, the form, and the template like in the previous recipes, *Filtering object lists* and *Managing paginated lists*.

How to do it...

We will need to create a URL rule in the URL configuration and add a class-based view. To include a class-based view in the URL rules, the `as_view()` method is used like this:

```
#movies/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, url
from views import MovieListView
urlpatterns = patterns('',
    url(r'^$', MovieListView.as_view(), name="movie_list"),
)
```

Our class-based view, `MovieListView`, will overwrite the `get` and `post` methods of the `View` class, which are used to distinguish between requests by GET and POST. We will also add the `get_queryset_and_facets` and `get_page` methods to make the class more modular:

```
#movies/views.py
# -*- coding: UTF-8 -*-
from django.shortcuts import render
from django.core.paginator import Paginator, EmptyPage,\
    PageNotAnInteger
from django.views.generic import View

from models import Genre
from models import Director
from models import Actor
from models import Movie, RATING_CHOICES
from forms import MovieFilterForm

class MovieListView(View):
    form_class = MovieFilterForm
    template_name = 'movies/movie_list.html'
    paginate_by = 15

    def get(self, request, *args, **kwargs):
        form = self.form_class(data=request.REQUEST)
        qs, facets = self.get_queryset_and_facets(form)
        page = self.get_page(request, qs)
        context = {
            'form': form,
            'facets': facets,
            'object_list': page,
        }
        return render(request, self.template_name, context)

    def post(self, request, *args, **kwargs):
        return self.get(request, *args, **kwargs)

    def get_queryset_and_facets(self, form):
        qs = Movie.objects.order_by('title')

        facets = {
            'selected': {},
            'categories': {
                'genres': Genre.objects.all(),
```

```
        'directors': Director.objects.all(),
        'actors': Actor.objects.all(),
        'ratings': RATING_CHOICES,
    },
}
if form.is_valid():
    genre = form.cleaned_data['genre']
    if genre:
        facets['selected']['genre'] = genre
        qs = qs.filter(genres=genre).distinct()

    director = form.cleaned_data['director']
    if director:
        facets['selected']['director'] = director
        qs = qs.filter(directors=director).distinct()

    actor = form.cleaned_data['actor']
    if actor:
        facets['selected']['actor'] = actor
        qs = qs.filter(actors=actor).distinct()

    rating = form.cleaned_data['rating']
    if rating:
        facets['selected']['rating'] = (int(rating),
                                         dict(RATING_CHOICES)[int(rating)])
        qs = qs.filter(rating=rating).distinct()
    return qs, facets

def get_page(self, request, qs):
    paginator = Paginator(qs, self.paginate_by)

    page_number = request.GET.get('page')
    try:
        page = paginator.page(page_number)
    except PageNotAnInteger:
        # If page is not an integer, show first page.
        page = paginator.page(1)
    except EmptyPage:
        # If page is out of range, show last existing page.
        page = paginator.page(paginator.num_pages)
    return page
```

How it works...

No matter whether the request was called by the GET or POST methods, we want the view to act the same; so, the `post` method is just calling the `get` method in this view, passing all positional and named arguments.

These are the things happening in the `get` method:

- ▶ At first, we create the `form` object passing the `REQUEST` dictionary-like object to it. The `REQUEST` object contains all the query variables passed using the GET or POST methods.
- ▶ Then, the `form` is passed to the `get_queryset_and_facets` method, which respectively returns a tuple of two elements: the `QuerySet` and the `facets` dictionary.
- ▶ Then, the current request object and the `QuerySet` is passed to the `get_page` method, which returns the current page object.
- ▶ Lastly, we create a context dictionary and render the response.

There is more...

As you see, the `get`, `post`, and `get_page` methods are quite generic, so that we could create a generic class, `FilterableListView`, with those methods in the `utils` app. Then in any app, which needs a filterable list, we could create a view that extends `FilterableListView` and defines only the `form_class` and `template_name` attributes and the `get_queryset_and_facets` method. This is how class-based views work.

See also

- ▶ The *Filtering object lists* recipe
- ▶ The *Managing paginated lists* recipe

Generating PDF documents

Django views allow you to create much more than just HTML pages. You can generate files of any type. For example, you can create PDF documents for invoices, tickets, booking confirmations, or some other purposes. In this recipe, we will show you how to generate resumes (curriculum vitae) in PDF format out of the data from the database. We will be using the **Pisa xhtml2pdf** library, which is very practical as it allows you to use HTML templates to make PDF documents.

Getting ready

First of all, we need to install the Python libraries `reportlab` and `xhtml2pdf` in your virtual environment:

```
(myproject_env)$ pip install reportlab==2.4
(myproject_env)$ pip install xhtml2pdf
```

Then, let's create a `cv` app with a simple CV model with the `Experience` model attached to it through a foreign key. The CV model will have these fields: first name, last name, and e-mail. The `Experience` model will have these fields: the start date at a job, the end date at a job, company, position at that company, and skills gained:

```
#cv/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

class CV(models.Model):
    first_name = models.CharField(_("First name"), max_length=40)
    last_name = models.CharField(_("Last name"), max_length=40)
    email = models.EmailField(_("Email"))

    def __unicode__(self):
        return self.first_name + " " + self.last_name

class Experience(models.Model):
    cv = models.ForeignKey(CV)
    from_date = models.DateField(_("From"))
    till_date = models.DateField(_("Till"), null=True, blank=True)
    company = models.CharField(_("Company"), max_length=100)
    position = models.CharField(_("Position"), max_length=100)
    skills = models.TextField(_("Skills gained"), blank=True)

    def __unicode__(self):
        till = _("present")
        if self.till_date:
            till = self.till_date.strftime('%m/%Y')
        return _("%(from)s-%(till)s %(position)s at %(company)s") % {
            'from': self.from_date.strftime('%m/%Y'),
            'till': till,
            'position': self.position,
            'company': self.company,
        }

class Meta:
    ordering = ("-from_date",)
```

How to do it...

In the URL rules, let's create a rule for the view to download a PDF document of a resume by the ID of the CV model, as follows:

```
#cv/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, url

urlpatterns = patterns('cv.views',
    url(r'^(?P<cv_id>\d+)/pdf/$', 'download_cv_pdf',
        name='download_cv_pdf'),
)
```

Now let's create the `download_cv_pdf` view. This view renders an HTML template and then passes it to the PDF creator `pisaDocument`:

```
#cv/views.py
# -*- coding: UTF-8 -*-
try:
    from cStringIO import StringIO
except ImportError:
    from StringIO import StringIO
from xhtml2pdf import pisa

from django.conf import settings
from django.shortcuts import get_object_or_404
from django.template.loader import render_to_string
from django.http import HttpResponse

from cv.models import CV

def download_cv_pdf(request, cv_id):
    cv = get_object_or_404(CV, pk=cv_id)

    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; ' \
        'filename=%s_%s.pdf' % (cv.first_name, cv.last_name)

    html = render_to_string("cv/cv_pdf.html", {
        'cv': cv,
        'MEDIA_ROOT': settings.MEDIA_ROOT,
        'STATIC_ROOT': settings.STATIC_ROOT,
    })
```

```
pdf = pisa.pisaDocument(
    StringIO(html.encode("UTF-8")),
    response,
    encoding='UTF-8',
)

return response
```

At last, we create the template by which the document will be rendered, as follows:

```
{#templates/cv/cv_pdf.html#}
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8" />
    <title>My Title</title>
    <style type="text/css">
      @page {
        size: "A4";
        margin: 2.5cm 1.5cm 2.5cm 1.5cm;
        @frame footer {
          -pdf-frame-content: footerContent;
          bottom: 0cm;
          margin-left: 0cm;
          margin-right: 0cm;
          height: 1cm;
        }
      }
      #footerContent {
        color: #666;
        font-size: 10pt;
        text-align: center;
      }
      /* ... Other CSS Rules go here ... */
    </style>
  </head>
  <body>
    <div>
      <p class="h1">Curriculum Vitae</p>
      <table>
        <tr>
          <td><p><b>{{ cv.first_name }} {{ cv.last_name }}</b><br><br />
              Contact: {{ cv.email }}</p>
```

```

        </td>
        <td align="right">
            
        </td>
    </tr>
</table>

<p class="h2">Experience</p>
<table>
    {% for experience in cv.experience_set.all %}
        <tr>
            <td><p>{{ experience.from_date|date:"F Y" }} -
                {% if experience.till_date %}
                    {{ experience.till_date|date:"F Y" }}
                {% else %}
                    present
                {% endif %}<br />
                {{ experience.position }} at {{
experience.company }}</p>
            </td>
            <td><p><b>Skills gained</b><br>
                {{ experience.skills|linebreaksbr }}
            <br>
            <br>
            </p>
            </td>
        </tr>
    {% endfor %}
</table>
</div>
<pdf:nextpage>
<div>
    This is an empty page to make a paper plane.
</div>
<div id="footerContent">
    Document generated at {% now "Y-m-d" %} |
    Page <pdf:pagenumber> of <pdf:pagecount>
</div>
</body>
</html>


```


How it works...

Depending on the data entered into the database, the rendered PDF document might look like this:

Curriculum Vitae

John Doe
Contact: john.doe@example.com



Experience

December 2012 - present Worker at Jack in the Box Not So Fine Dining	Skills gained Contrary to popular belief, this place is not where they make Jack in the Box toys. Now I know why they asked me if I knew how to cook a hamburger in the interview!
August 2012 - present Sumo Wrestler at Wherever	Skills gained I pursued my dream of Sumo Wrestling, only to find out that the minimum weight limit was 300 pounds. Sadly, my Sumo career did not last but I have high hopes for returning to my passion in the near future.
March 2009 - March 2011 Local Taste Tester at Whole Foods	Skills gained Every day I went to my local Whole Foods and did quality control on all samples. Eventually it was found that I was in fact not on the payroll, so I was escorted from the building. I even worked weekends! Fortunately, the restraining order will be lifted soon and I will be able to return from my hiatus.
January 2004 - March 2009 Babysitter at Home	Skills gained I was forced to babysit my little brother for years while my parents worked. He used to cry constantly, but I guess I would too if my older brother sat on me.

Document generated at 2014-04-04 | Page 1 of 2

How does the view work? At first, we load a curriculum vitae by its ID if it exists, or raise the `Page-not-found` error if it doesn't. Then, we create the response object with `mimetype` of the PDF document. We set the `Content-Disposition` header to `attachment` with the specified filename. This will force browsers to open a dialog box prompting to save the PDF document and will suggest the specified name for the file. Then, we render the HTML template as a string passing curriculum vitae object, and the `MEDIA_ROOT` and `STATIC_ROOT` paths.



Note that the `src` attribute of the `` tag used for PDF creation needs to point to the file in the filesystem or the full URL of the image online.

Then, we create a `pisaDocument` file with the UTF-8 encoded HTML as source, and response object as the destination. The response object is a file-like object, and `pisaDocument` writes the content of the document to it. The response object is returned by the view as expected.

Let's have a look at the HTML template used to create this document. The template has some unusual HTML tags and CSS rules. If we want to have some elements on each page of the document, we can create CSS frames for that. In the preceding example, the `<div>` tag with the `footerContent` ID is marked as a frame, which will be repeated at the bottom of each page. In a similar way, we can have a header or a background image for each page.

Here are the specific HTML tags used in this document:

- ▶ The `<pdf:nextpage>` tag sets a manual page break
- ▶ The `<pdf:pagenumber>` tag returns the number of the current page
- ▶ The `<pdf:pagecount>` tag returns the total number of pages

The current version 3.0.33 of the Pisa `xhtml2pdf` library doesn't fully support all HTML tags and CSS rules; for example, `<h1>`, `<h2>`, and other headings are broken and there is no concept of floating, so instead you have to use paragraphs with CSS classes for different font styles and tables for multi-column layouts. However, this library is still mighty enough for customized layouts, which basically can be created just with the knowledge of HTML and CSS.

See also

- ▶ The *Managing paginated lists* recipe

4

Templates and JavaScript

In this chapter, we will cover the following topics:

- ▶ Arranging the base.html template
- ▶ Including JavaScript settings
- ▶ Using HTML5 data attributes
- ▶ Opening object details in a pop up
- ▶ Implementing a continuous scroll
- ▶ Implementing the Like widget
- ▶ Uploading images by Ajax

Introduction

We are living in the Web2.0 world where social web applications and smart websites communicate between servers and clients using Ajax, refreshing whole pages only when the context changes. In this chapter, you will learn best practices to deal with JavaScript in your templates in order to create a rich user experience. For responsive layouts, we will use the Bootstrap 3 frontend framework. For efficient scripting, we will use the jQuery JavaScript framework.

Arranging the base.html template

When you start working on templates, one of the first actions is to create the boilerplate `base.html`, which will be extended by most of the page templates in your project. In this recipe, I will demonstrate how to create such a template for multilingual HTML5 websites with responsiveness in mind.



Responsive websites are those that adapt to the screen size of the device whether the visitor uses desktop browsers, tablets, or phones.

Getting ready

Create the template's directory in your project and set `TEMPLATE_DIRS` in `settings.py`.

How to do it...

Perform the following steps:

1. In the root directory of your template, create a `base.html` file with the following content:

```
{#templates/base.html#}  
{% block doctype %}<!DOCTYPE html>{% endblock %}  
{% load i18n %}  
<html lang="{{ LANGUAGE_CODE }}">  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width,  
initial-scale=1" />  
    <title>{% block title %}{% endblock %}{% trans "My Website"  
%}</title>  
    <link rel="icon" href="{{ STATIC_URL }}site/img/favicon.ico"  
type="image/png" />  
  
    {% block meta_tags %}{% endblock %}  
  
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.  
com/bootstrap/3.1.1/css/bootstrap.min.css" />  
    <link href="{{ STATIC_URL }}site/css/style.css"  
rel="stylesheet" media="screen" type="text/css" />  
  
    {% block stylesheet %}{% endblock %}
```

```

<script src="http://code.jquery.com/jquery-1.11.0.min.js"></
script>
  <script src="http://code.jquery.com/jquery-migrate-1.2.1.min.
js"></script>
  <script src="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/
js/bootstrap.min.js"></script>
  <script src="{% url 'js_settings' %}"></script>

  {% block js %}{% endblock %}
  {% block extrahead %}{% endblock %}
</head>
<body class="{% block bodyclass %}{% endblock %}">
  {% block page %}
    <div class="wrapper">
      <div id="header" class="clearfix">
        <h1>{% trans "My Website" %}</h1>
        {% block header_navigation %}
          {% include "utils/header_navigation.html" %}
        {% endblock %}
        {% block language_chooser %}
          {% include "utils/language_chooser.html" %}
        {% endblock %}
      </div>
      <div id="content" class="clearfix">
        {% block content %}
        {% endblock %}
      </div>
      <div id="footer" class="clearfix">
        {% block footer_navigation %}
          {% include "utils/footer_navigation.html" %}
        {% endblock %}
      </div>
    </div>
  {% endblock %}
  {% block extrabody %}{% endblock %}
</body>
</html>

```

2. In the same directory, create another file named `base_simple.html` for specific cases:

```
{# templates/base_simple.html #}
{% extends "base.html" %}

{% block page %}
    <div class="wrapper">
        <div id="content" class="clearfix">
            {% block content %}
            {% endblock %}
        </div>
    </div>
{% endblock %}
```

How it works...

The base template contains the `<head>` and `<body>` sections of the HTML document, with all the details that are reused on each page of the website. Depending on the web design requirements, you can have additional base templates for different layouts. For example, we added the `base_simple.html` file, which has the same HTML `<head>` section and very minimalistic `<body>` section, and it can be used for login screen, password reset, or other simple pages. You can have separate base templates for single-column, two-column, and three-column layouts, where each of them extends `base.html` and overwrites the content of the `<body>` section.

Let's look into the details of the `base.html` template we defined earlier. The doctype can be overwritten in a specific template if, for some reason, you need a different doctype, or a custom **Document Type Definition** for one of the pages.

In the `<head>` section, we define UTF-8 as the default encoding to support multilingual content. Then, we have the viewport definition that will scale the website in the browser to use the full width. This is necessary for small-screen devices that will get specific screen layouts created with the Bootstrap frontend framework. Of course, there is a customizable website title, and the favicon will be shown in the browser's tab. We have extendable blocks for meta tags, style sheets, JavaScript, and whatever else that might be necessary in the `<head>` section. Note that we load Bootstrap styles and JavaScript in the template because we want to have responsive layouts and basic solid predefined styles for all elements. Then, we load the jQuery JavaScript library that allows us to create rich user experiences efficiently and flexibly. We also load JavaScript settings that are rendered from a Django view. You will learn about this in the next recipe.

In the `<body>` section, we have the header with an overwriteable navigation and a language chooser. We also have the content block and footer. At the very bottom, there is an extendable block for additional markup or JavaScript.

The base template we created is by no means a static unchangeable template. You can add to it whatever elements you need, for example, Google Analytics code, common JavaScript files, the Apple touch icon for iPhone bookmarks, Open Graph meta tags, Twitter Card tags, and so on.

See also

- The *Including JavaScript settings* recipe

Including JavaScript settings

In Django projects, we set different configuration values in the `settings.py` file. Some of these values also need to be set in JavaScript. As we want a single location to define our project settings, and we don't want to repeat the process when setting the configuration for the JavaScript values, it is good practice to include a dynamically generated configuration file into the base template. In this recipe, I will show you how to do that.

Getting ready

Make sure you have the `media`, `static`, and `request` context processors set for the `TEMPLATE_CONTEXT_PROCESSORS` setting:

```
#settings.py
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.tz",
    "django.contrib.messages.context_processors.messages",
    "django.core.context_processors.request",
)
```

Also, create the `utils` app if you haven't done so already, and place it under `INSTALLED_APPS` in the settings.

How to do it...

Follow these steps to create and include JavaScript settings:

1. Create a URL rule to call a view that renders JavaScript settings, as follows:

```
#urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, include, url
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns("",
    # ...
    url(r"^js-settings/$", "utils.views.render_js",
        {"template_name": 'settings.js'}, name="js_settings"),
)
```

2. In the views of your utils app, create the `render_js` view that returns a response of the JavaScript content type, as follows:

```
#utils/views.py
# -*- coding: utf-8 -*-
from datetime import datetime, timedelta
from django.shortcuts import render
from django.views.decorators.cache import cache_control

@cache_control(public=True)
def render_js(request, cache=True, *args, **kwargs):
    response = render(request, *args, **kwargs)
    response["Content-Type"] = \
        "application/javascript; charset=UTF-8"
    if cache:
        now = datetime.utcnow()
        response["Last-Modified"] = \
            now.strftime("%a, %d %b %Y %H:%M:%S GMT")
        # cache in the browser for 1 month
        expires = now + timedelta(days=31)

        response["Expires"] = \
            expires.strftime("%a, %d %b %Y %H:%M:%S GMT")
    else:
        response["Pragma"] = "No-Cache"
    return response
```

3. Create a `settings.js` template that returns JavaScript with the global settings JavaScript object, as follows:

```
#templates/settings.js
window.settings = {
    MEDIA_URL: '{{ MEDIA_URL|escapejs }}',
    STATIC_URL: '{{ STATIC_URL|escapejs }}',
    lang: '{{ LANGUAGE_CODE|escapejs }}',
    languages: { {% for lang_code, lang_name in LANGUAGES %}'{{
lang_code|escapejs }}': '{{ lang_name|escapejs }}' {% if not
forloop.last %},{% endif %} {% endfor %} }
};
```

4. Finally, include the rendered JavaScript settings file into the base template, as follows:

```
#templates/base.html
<script src="{% url "js_settings" %}"></script>
```

How it works...

The Django template system is very flexible; you are not limited to using templates just for HTML. In this example, we dynamically create the JavaScript file, the content of which will be something like this:

```
window.settings = {
    MEDIA_URL: '/media/',
    STATIC_URL: '/static/20140424140000/',
    lang: 'en',
    languages: { 'en': 'English', 'de': 'Deutsch', 'fr': 'Français',
'lt': 'Lietuvių kalba' }
};
```

The view will be cacheable both in the server and browser.

If you want to pass more variables to the JavaScript settings, either create a custom view and pass all the values to the context or create a custom context processor and pass all the values there. In the latter case, the variables will also be accessed in all the templates of your project. For example, you might have indicators such as `{{ is_mobile }}`, `{{ is_tablet }}`, and `{{ is_desktop }}` in your templates, telling the User-Agent string whether the visitor uses a mobile, tablet, or desktop browser.

See also

- ▶ The *Arranging the base.html template* recipe
- ▶ The *Using HTML5 data attributes* recipe

Using HTML5 data attributes

When you have dynamic data related to DOM elements, you need a more efficient way to pass values from Django to JavaScript. In this recipe, I will show you a way to attach data from Django to custom HTML5 data attributes, and then describe how to read the data from JavaScript in two practical examples. The first example will be an image that changes its source depending on the screen size so that the smallest version is shown on mobile devices, the medium-sized version is shown on tablets, and the biggest high-quality image is shown for the desktop version of the website. The second example will be a Google Map with a marker at a specified geographical position.

Getting ready

To get started, perform the following steps:

1. Create a `locations` app with a model `Location`, which will have at least the `title` character field, the `small_image`, `medium_image`, and `large_image` image fields, and the `latitude` and `longitude` floating-point fields.
2. Create an administration for this model and enter a sample location.
3. Lastly, create a detailed view for the location and set the URL rule for it.

How to do it...

Perform the following steps:

1. As we have the app created, we will need the template for the location detail, as follows:

```
{% templates/locations/location_detail.html#}
{% extends "base.html" %}

{% block content %}
    <h2>{{ location.title }}</h2>

    
```

```

    <div id="map"
        data-latitude="{{ location.latitude|stringformat:"f" }}"
        data-longitude="{{ location.longitude|stringformat:"f"
    }}"
    ></div>
{% endblock %}

{% block extrabody %}
    <script type="text/JavaScript" src="https://maps-api-ssl.
google.com/maps/api/js?v=3&sensor=true"></script>
    <script src="{{ STATIC_URL }}site/js/location.js"></script>
{% endblock %}

```

2. Besides the template, we need the JavaScript file that will read out the HTML5 data attributes and use them accordingly, as follows:

```

//site_static/site/js/location.js
function show_best_images() {
    $('img.img-full-width').each(function() {
        var $img = $(this);
        if ($img.width() > 1024) {
            $img.attr('src', $img.data('large-src'));
        } else if ($img.width() > 468) {
            $img.attr('src', $img.data('medium-src'));
        } else {
            $img.attr('src', $img.data('small-src'));
        }
    });
}

function show_map() {
    var $map = $('#map');
    var latitude = parseFloat($map.data('latitude'));
    var longitude = parseFloat($map.data('longitude'));
    var latlng = new google.maps.LatLng(latitude, longitude);

    var map = new google.maps.Map($map.get(0), {
        zoom: 15,
        center: latlng
    });
    var marker = new google.maps.Marker({
        position: latlng,
        map: map
    });
}

```

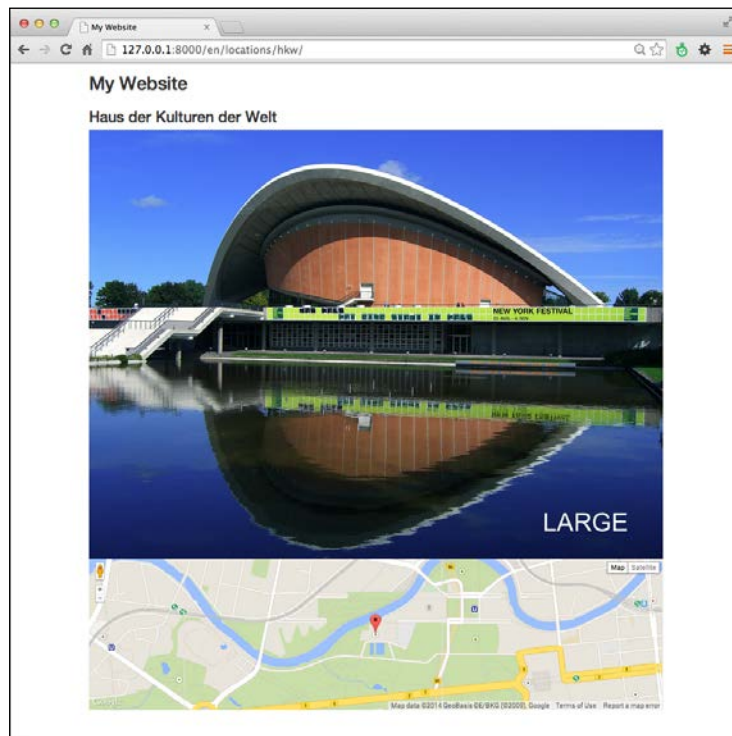
```
$(document).ready(function() {  
    show_best_images();  
    show_map();  
});  
  
$(window).on('resize', show_best_images);
```

3. Finally, we need to set some CSS, as follows:

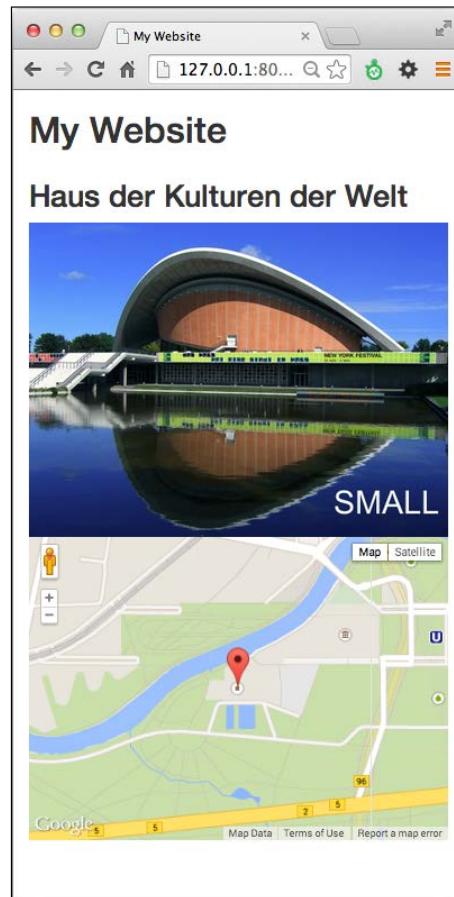
```
/*site_static/site/css/style.css*/  
img.img-full-width {  
    width: 100%;  
}  
#map {  
    height: 300px;  
}
```

How it works...

If you open your location detail view in a browser, you will see something like this in the large window:



If you resize the browser window to the narrowest size possible, the image will change to the smallest version, as shown here:



Let's take a look at the code. In the template, we have an image tag with a CSS class, `img-full-width`, and custom attributes `data-small-src`, `data-medium-src`, and `data-large-src`. In the JavaScript, the `show_best_images` function is called to load a page and resize a window. The function goes through all images with the CSS `img-full-width` class and sets appropriate image sources from the custom data attributes, depending on the current image width.

Then, there is a `<div>` element with the `map` ID and the `data-latitude` and `data-longitude` custom attributes in the template. In the JavaScript, a function named `show_map` is called on page load. This function will create a Google Map inside of the `<div>` element. At first, the custom attributes are read and converted from strings to floating-point values. Then, the `LatLng` object is created, which, in the next steps, becomes the center of the map and the geographical position of the marker shown on that map.

See also

- ▶ The *Including JavaScript settings* recipe
- ▶ The *Opening object details in a pop up* recipe

Opening object details in a pop up

In this recipe, we will create a list of links to locations, which when clicked, open a Bootstrap 3 modal dialog (or a pop-up window) with some information about the location and more links leading to the location detail page. The content for the dialog will be loaded by Ajax. For visitors without JavaScript, the detail page will open immediately, without this intermediate step.

Getting ready

Let's start with the locations app that we created in the previous recipe.

In the `urls.py` file, we will have three rules: one for the location list, one for the location detail, and one for the dialog, as follows:

```
#locations/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, url

urlpatterns = patterns("locations.views",
    url(r"'^$', "location_list", name="location_list"),
    url(r"^(?P<slug>[^/]+)/$", "location_detail",
        name="location_detail"),
    url(r"^(?P<slug>[^/]+)/popup/$", "location_detail_popup",
        name="location_detail_popup"),
)
```

Consequently, there will be three simple views, as follows:

```
#locations/views.py
# -*- coding: UTF-8 -*-
from django.shortcuts import render, get_object_or_404
from models import Location

def location_list(request):
    location_list = Location.objects.all()
    return render(request, "locations/location_list.html",
        {"location_list": location_list})
```

```
def location_detail(request, slug):
    location = get_object_or_404(Location, slug=slug)
    return render(request, "locations/location_detail.html",
                  {"location": location})

def location_detail_popup(request, slug):
    location = get_object_or_404(Location, slug=slug)
    return render(request, "locations/location_detail_popup.html",
                  {"location": location})
```

How to do it...

Execute these steps one by one:

1. Create a template for the location's list view with a hidden empty modal dialog at the end. Each listed location will have custom HTML5 data attributes dealing with the pop-up information, as follows:

```
{#templates/locations/location_list.html#}
{% extends "base.html" %}
{% load i18n %}

{% block content %}
    <h2>{% trans "Locations" %}</h2>
    <ul>
        {% for location in location_list %}
            <li class="item">
                <a href="{% url "location_detail" slug=location.
slug %}"
                    data-popup-url="{% url "location_detail_popup"
slug=location.slug %}"
                    data-popup-title="{%{{ location.title|escape %}}">
                        {%{{ location.title %}}
                    </a>
            </li>
        {% endfor %}
    </ul>
{% endblock %}

{% block extrabody %}
    <div id="popup" class="modal fade">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
```



```

        <button type="button" class="close" data-
dismiss="modal" aria-hidden="true">&times;</button>
        <h4 class="modal-title">Modal title</h4>
    </div>
    <div class="modal-body">
    </div>
</div>
</div>
</div>
<script src="{{ STATIC_URL }}site/js/location_list.js"></
script>
{% endblock %}

```

2. We need JavaScript to handle the opening of the dialog and loading content dynamically, as follows:

```

// site_static/site/js/location_list.js
$(document).ready(function() {
    var $popup = $('#popup');

    $('a.item').click(function(){
        var $link = $(this);
        var popup_url = $link.data('popup-url');
        var popup_title = $link.data('popup-title');

        if (!popup_url) {
            return true;
        }
        $('.modal-title', $popup).html(popup_title);
        $('.modal-body', $popup).load(popup_url, function() {
            $popup.on('shown.bs.modal', function () {
                // do something when dialog is shown
            }).modal("show");
        });

        $('.close', $popup).click(function() {
            // do something when dialog is closing
        });

        return false; // disable default link functionality
    });
});

```

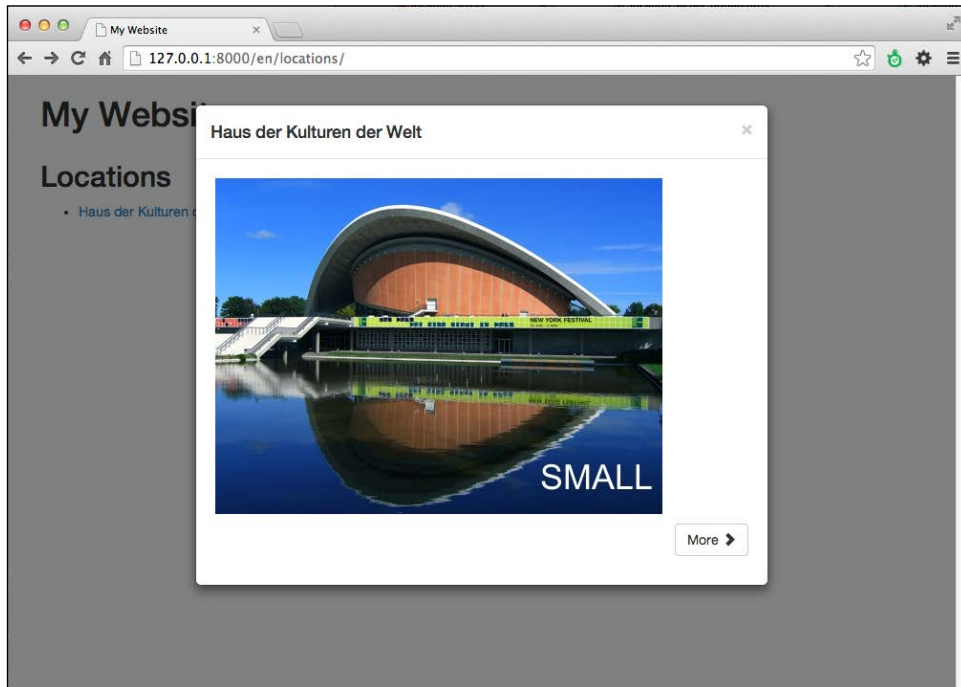
- Finally, we create a template for the content that will be loaded into the modal dialog, as follows:

```
{# templates/locations/location_detail_popup.html #}
{% load i18n %}
<p></p>

<p class="clearfix">
  <a href="{% url "location_detail" slug=location.slug %}"
    class="btn btn-default pull-right">
    {% trans "More" %}
    <span class="glyphicon glyphicon-chevron-right"></span>
  </a>
</p>
```

How it works...

If we go to the location's list view in a browser and click on one of the locations, we will see a pop up like this:



How does this work? In the template, there is a `<div>` element with the "item" CSS class and a link for each location. The links have the `data-popup-url` and `data-popup-title` custom attributes. In the JavaScript, on the page load for each link within an element with the "item" CSS class, we assign an `onclick` event handler that reads custom attributes as `popup_url` and `popup_title`, sets the new title to the hidden dialog box, loads the content to the pop up using Ajax, and then shows it to the visitor.

See also

- ▶ The *Using HTML5 data attributes* recipe
- ▶ The *Implementing a continuous scroll* recipe
- ▶ The *Implementing the Like widget* recipe

Implementing a continuous scroll

Social websites often have the feature of continuous scrolling, which is also known as infinite scrolling; there are long lists of items, and as you scroll the page down, new items are loaded and attached to the bottom automatically. In this recipe, I will show you how to achieve such an effect with Django and the jScroll jQuery plugin. We'll illustrate this using a sample view showing The Top 250 movies of all time from Internet Movie Database (<http://www.imdb.com/>).

Getting ready

First, download the jScroll plugin from the following link:

<https://github.com/pklauzinski/jscroll>

Next, for this example, you will create a `movies` app with a paginated list view for the movies. You can either create a `Movie` model, or a list of dictionaries with movie data. Every movie will have the place in the TOP, title, release year, and rating fields.

How to do it...

Perform the following steps:

1. The first step is to create a template for the list view, which will also show a link to the next page, as follows:

```
{#templates/movies/movie_list.html#}  
{% extends "base.html" %}  
{% load i18n utility_tags %}
```

```

{% block content %}
    <h2>{% trans "Top Movies" %}</h2>
    <div class="object_list">
        {% for movie in object_list %}
            <div class="item">
                <p>{{ movie.place }}.
                    <strong>{{ movie.title }}</strong>
                    ({{ movie.year }})
                    <span class="badge">{% trans "IMDB rating" %}:
{{ movie.rating }}</span>
                </p>
            </div>
        {% endfor %}
        {% if object_list.has_next %}
            <p class="pagination"><a class="next_page" href="{%
append_to_query page=object_list.next_page_number %}">{% trans
"More..." %}</a></p>
        {% endif %}
    </div>
{% endblock %}

{% block extrabody %}
    <script src="{{ STATIC_URL }}site/js/jquery.jscroll.min.js"></
script>
    <script src="{{ STATIC_URL }}site/js/list.js"></script>
{% endblock %}

```

2. The second step is to add some JavaScript, as follows:

```

// site_static/site/js/list.js
$(document).ready(function() {
    $('.object_list').jscroll({
        loadingHtml: '',
        padding: 100,
        pagingSelector: '.pagination',
        nextSelector: 'a.next_page:last',
        contentSelector: '.item,.pagination'
    });
});

```

How it works...

When you open the movie-list view in a browser, a predefined number of items, for example, 25, is shown on the page. As you scroll down, an additional 25 items and the next pagination link are loaded and appended to the item container. Then the third page of items is loaded and attached to the bottom, and this continues until there are no more pages.

Upon page load, the `<div>` tag in the JavaScript, which has the `"object_list"` CSS class and contains the items and pagination links, will become a `jScroll` object. The following are the parameters defining its features:

- ▶ `loadingHtml`: This sets an animated loading indicator shown at the end of the list when a new page is loading
- ▶ `padding`: The new page will be triggered to load when there are 100 pixels between the scrolling position and the end of the scrolling area
- ▶ `pagingSelector`: HTML elements found by this CSS selector will be hidden in browsers, with JavaScript switched on
- ▶ `nextSelector`: HTML elements found by this CSS selector will be used to read the URL of the next page
- ▶ `contentSelector`: This CSS selector defines HTML elements to take out of the loaded content and put into the container

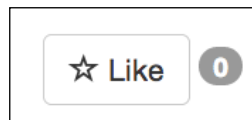
See also

- ▶ The *Managing paginated lists* recipe in *Chapter 3, Forms and Views*
- ▶ The *Composing class-based views* recipe in *Chapter 3, Forms and Views*
- ▶ The *Including JavaScript settings* recipe

Implementing the Like widget

Nowadays, social websites usually have integrated Facebook, Twitter, and Google+ widgets to like and share pages. In this recipe, I will guide you through a similar internal liking Django app that saves all likes in your database so that you can create specific views based on the things liked on your website. We will create a `Like` widget with a two-state button and a badge showing the number of total likes. These are the states:

- ▶ Inactive state, where you can click on a button to activate it:



- ▶ Active state, where you can click on a button to deactivate it:



The states of the widget will be handled by Ajax calls.

Getting ready

First, create a `likes` app with a `Like` model, which has a foreign-key relation to the user who is **liking** something and a generic relationship to any object in the database. We will use `ObjectRelationMixin`, which we defined in the *Creating a model mixin to handle generic relations* recipe in *Chapter 2, Database Structure*. If you don't want to use the mixin, you can also define a generic relation in the following model:

```
#likes/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.conf import settings
from utils.models import ObjectRelationMixin
from utils.models import CreationModificationDateMixin

class Like(CreationModificationDateMixin,
            ObjectRelationMixin(is_required=True)):
    user = models.ForeignKey(settings.AUTH_USER_MODEL)

    class Meta:
        verbose_name = _("like")
        verbose_name_plural = _("likes")
        ordering = ("-created",)

    def __unicode__(self):
        return _(u"%(user)s likes %(obj)s") % {
            "user": self.user,
            "obj": self.content_object,
        }
```

Also, make sure that the `request` context processor is set in the settings. We also need an authentication middleware in the settings for the currently logged-in user attached to the request:

```
#settings.py
TEMPLATE_CONTEXT_PROCESSORS = (
    # ...
    "django.core.context_processors.request",
)
MIDDLEWARE_CLASSES = (
    # ...
    "django.contrib.auth.middleware.AuthenticationMiddleware",
)
```

How to do it...

Execute these steps one by one:

1. Inside the `likes` app, create a `templatetags` directory with an empty file, `__init__.py`, to make it a Python module. Then, add the `likes_tags.py` file, where we'll define the `{% like_widget %}` template tag, as follows:

```
#likes/templatetags/likes_tags.py
# -*- coding: UTF-8 -*-
from django import template
from django.contrib.contenttypes.models import ContentType
from django.template import loader

from likes.models import Like

register = template.Library()

### TAGS ###

@register.tag
def like_widget(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, for_str, obj = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError, \
            "%r tag requires a following syntax: " \
            "{%% %r for <object> %%}" % (
                token.contents[0], token.contents[0])
    return ObjectLikeWidget(obj)
```

```

class ObjectLikeWidget(template.Node):
    def __init__(self, obj):
        self.obj = obj

    def render(self, context):
        obj = template.resolve_variable(self.obj, context)
        ct = ContentType.objects.get_for_model(obj)

        is_liked_by_user = bool(Like.objects.filter(
            user=context["request"].user,
            content_type=ct,
            object_id=obj.pk,
        ))

        context.push()
        context["object"] = obj
        context["content_type_id"] = ct.pk
        context["is_liked_by_user"] = is_liked_by_user
        context["count"] = get_likes_count(obj)

        output = loader.render_to_string(
            "likes/includes/like.html", context)
        context.pop()
        return output

```

2. Also, we'll add a filter in the same file to get the number of likes for a specified object:

```

### FILTERS ###

@register.filter
def get_likes_count(obj):
    ct = ContentType.objects.get_for_model(obj)
    return Like.objects.filter(
        content_type=ct,
        object_id=obj.pk,
    ).count()

```

3. In the URL rules, we need a rule for a view, which will handle the **liking** and **unliking** using Ajax:

```

#likes/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, url

urlpatterns = patterns("likes.views",

```



```
url(r"^(?P<content_type_id>[/]+)/(?P<object_id>[/]+)/$",
    "json_set_like", name="json_set_like"),
)
```

4. Then, we need to define the view, as follows:

```
#likes/views.py
# -*- coding: UTF-8 -*-
import json
from django.http import HttpResponse
from django.views.decorators.cache import never_cache
from django.contrib.contenttypes.models import ContentType
from django.shortcuts import render
from django.views.decorators.csrf import csrf_exempt

from likes.models import Like
from likes.templatetags.likes_tags import get_likes_count

@never_cache
@csrf_exempt
def json_set_like(request, content_type_id, object_id):
    """
    Sets the object as a favorite for the current user
    """
    json_str = "false"
    if request.user.is_authenticated() and \
        request.method == "POST":
        content_type = ContentType.objects.get(id=content_type_id)
        obj = content_type.get_object_for_this_type(pk=object_id)
        like, is_created = Like.objects.get_or_create(
            content_type=ContentType.objects.get_for_model(obj),
            object_id=obj.pk,
            user=request.user,
        )
        if not is_created:
            like.delete()
        result = {
            "obj": unicode(obj),
            "action": is_created and "added" or "removed",
            "count": get_likes_count(obj),
        }
        json_str = json.dumps(result, ensure_ascii=False,
                               encoding="utf8")
        return HttpResponse(json_str,
                             mimetype="text/javascript; charset=utf-8")
```

5. In the template for the list or detail view of any object, we can add the template tag for the widget. Let's add the widget to the location detail that we created in the previous recipes:

```
{# templates/locations/location_detail.html #}
{% extends "base.html" %}
{% load likes_tags %}

{% block content %}
    {% if request.user.is_authenticated %}
        {% like_widget for location %}
    {% endif %}
    {# the details of the object go here... #}
{% endblock %}

{% block extrabody %}
    <script src="{ { STATIC_URL } }site/js/likes.js"></script>
{% endblock %}
```

6. Then, we need a template for the widget, as follows:

```
{#templates/likes/includes/like.html#}
{% load i18n %}
<div class="like-widget">
    <button class="like-button btn btn-default {% if is_liked_by_
user %} active{% endif %}"
        data-href="{% url "json_set_like" content_type_id=content_
type_id object_id=object.pk %}"
        data-like-text="{% trans "Like" %}"
        data-unlike-text="{% trans "Unlike" %}"
    >
        {% if is_liked_by_user %}
            <span class="glyphicon glyphicon-star"></span>
            {% trans "Unlike" %}
        {% else %}
            <span class="glyphicon glyphicon-star-empty"></span>
            {% trans "Like" %}
        {% endif %}
    </button>
    <span class="like-badge badge">{{ count }}</span>
</div>
```

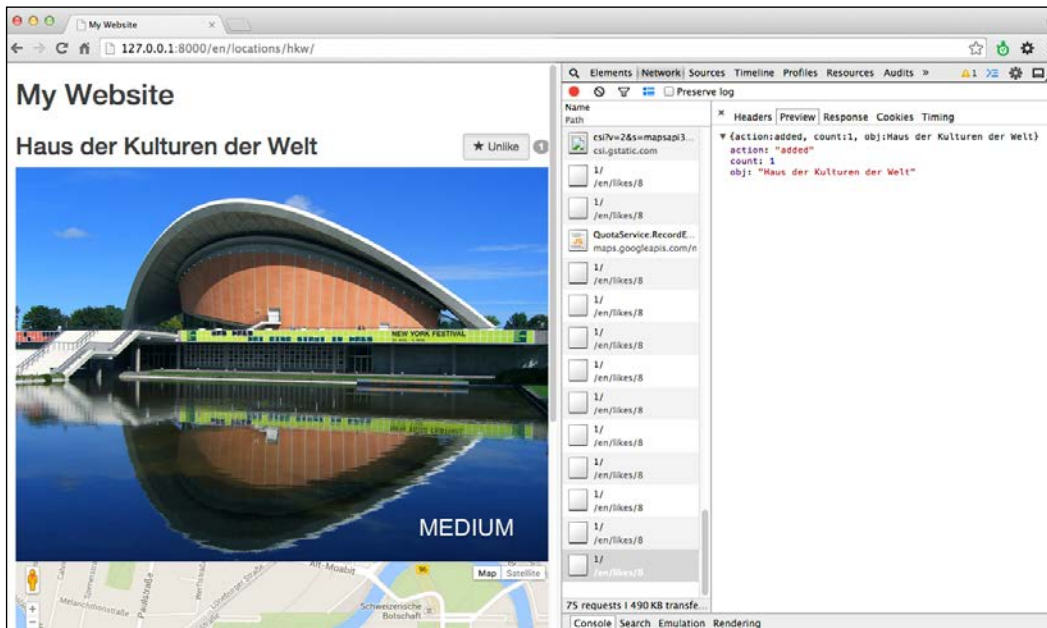
7. Finally, we create JavaScript to handle **liking** and **unliking** in the browser, as follows:

```
// site_static/site/js/likes.js
$(document).on('click', '.like-button', function() {
    var $button = $(this);
    var $badge = $button.closest('.like-widget').find('.like-
badge');
    $.post($button.data('href'), function(data) {
        if (data['action'] == 'added') {
            $button.addClass('active').html('<span
class="glyphicon glyphicon-star"></span> ' + $button.data('unlike-
text'));
        } else {
            $button.removeClass('active').html('<span
class="glyphicon glyphicon-star-empty"></span> ' + $button.
data('like-text'));
        }
        $badge.html(data['count']);
    }, 'json');
});
```

How it works...

For any object in your website, you can put the `{% like_widget for object %}` template tag that will check if the object is already liked and will show an appropriate state. The custom HTML5 attributes, `data-href`, `data-like-text`, and `data-unlike-text` are in the widget template. The first attribute holds a unique object-specific URL to change the current state of the widget. The other two attributes hold the translated texts for the widget. In the JavaScript, liking buttons are recognized by the `"like-button"` CSS class. For each such button, there is an `onclick` handler attached that posts an Ajax call to the URL specified by the `data-href` attribute. The specified view accepts two of the parameters, content type and object ID, of the liked object. The view checks whether a `Like` for the specified object exists, and if it does, the view removes it, otherwise the `Like` object is added. As a result, the view returns a JSON response with the liked object's text representation, actions whether the `Like` object was added or removed, and counts the total number of likes. Depending on the action returned, JavaScript will show an appropriate state for the button.

You can debug the Ajax responses in Chrome Developer Tools or the Firefox Firebug plugin. If any server errors occur while developing, you will see the error trace back in the preview of the response:



See also

- ▶ The *Opening object details in a pop up* recipe
- ▶ The *Implementing a continuous scroll* recipe
- ▶ The *Uploading images by Ajax* recipe
- ▶ The *Creating a model mixin to handle generic relations* recipe in *Chapter 2, Database Structure*
- ▶ *Chapter 5, Custom Template Filters and Tags*

Uploading images by Ajax

Ajaxified file uploads have become the de facto standard on the web. People want to see what they choose right after selecting a file instead of after saving a form. Also, if the form has validation errors, nobody wants to select the files again; the file should still be selected in the form with validation errors.

There is a third-party app, **django-ajax-uploader**, which can be used to upload images by Ajax. In this recipe, I will show you how to do this.

Getting ready

Let's start with the quotes app that we created for the *Uploading images* recipe in *Chapter 3, Forms and Views*. We will reuse the model and view, but we'll create a different form and template, and will add some JavaScript too.

Install **django-crispy-forms** and **django-ajax-uploader** to your local environment using the following commands:

```
(myproject)$ pip install django-crispy-forms
(myproject)$ pip install ajaxuploader
```

Don't forget to put these apps into `INSTALLED_APPS`:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "quotes",
    "crispy_forms",
    "ajaxuploader",
)
```

How to do it...

Let's redefine the form for inspirational quotes using the following steps:

1. First, we create a layout for the Bootstrap 3 markup. Note that instead of the `picture` image field, we have the hidden fields, `picture_path` and `delete_picture`, and some markup for the file-upload widget:

```
#quotes/forms.py
# -*- coding: UTF-8 -*-
import os
from django import forms
from django.utils.translation import ugettext_lazy as _
```

```

from django.core.files import File
from django.conf import settings
from crispy_forms.helper import FormHelper
from crispy_forms import layout, bootstrap
from models import InspirationQuote

class InspirationQuoteForm(forms.ModelForm):
    picture_path = forms.CharField(
        max_length=255,
        widget=forms.HiddenInput(),
        required=False,
    )
    delete_picture = forms.BooleanField(
        widget=forms.HiddenInput(),
        required=False,
    )

    class Meta:
        model = InspirationQuote
        fields = ["author", "quote"]

    def __init__(self, *args, **kwargs):
        super(InspirationQuoteForm, self).\
            __init__(*args, **kwargs)

        self.helper = FormHelper()
        self.helper.form_action = ""
        self.helper.form_method = "POST"

        self.helper.layout = layout.Layout(
            layout.Fieldset(
                _("Quote"),
                layout.Field("author"),
                layout.Field("quote", rows=3),
                layout.HTML(u""{% load i18n %}
                    <div id="image_upload_widget">
                        <div class="preview">
                            {% if instance.picture %}
                                

                                {% endif %}
                        </div>
                        <div class="uploader">
                            <noscript>

```

```

                                <p>{% trans "Please enable
JavaScript to use file uploader." %}</p>
                                </noscript>
                                </div>
                                <p class="help_text" class="help-block">
                                    {% trans "Available formats are JPG,
GIF, and PNG." %}
                                </p>
                                <div class="messages"></div>
                            </div>
                            """),
                            layout.Field("picture_path"),
                            layout.Field("delete_picture"),
                        ),
                        bootstrap.FormActions(
                            layout.Submit("submit", _("Save"),
                                css_class="btn btn-primary"),
                        )
                    )
                )

```

2. Then, we overwrite the save method to handle the saving of the inspiration quote, as follows:

```

def save(self, commit=True):
    instance = super(InspirationQuoteForm, self).\
        save(commit=True)

    if self.cleaned_data["delete_picture"] and \
        instance.picture:
        instance.picture.delete()

    if self.cleaned_data["picture_path"]:
        tmp_path = self.cleaned_data['picture_path']
        abs_tmp_path = os.path.join(settings.MEDIA_ROOT,
            tmp_path)
        filename = InspirationQuote._meta.\
            get_field("picture").upload_to(instance, tmp_path)
        instance.picture.save(filename,
            File(open(abs_tmp_path, "rb")), False)

        os.remove(abs_tmp_path)
    instance.save()
    return instance

```

3. In addition to the previously defined views in the quotes app, we add the `ajax_uploader` view that will handle uploads by Ajax, as follows:

```
#quotes/views.py
# ...
from ajaxuploader.views import AjaxFileUploader
ajax_uploader = AjaxFileUploader()
```

4. Then, we set the URL rule for the view, as follows:

```
#quotes/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, url

urlpatterns = patterns("",
    # ...
    url(r'^ajax-upload/$', "quotes.views.ajax_uploader",
        name="ajax_uploader"),
)
```

5. Then, it is time to create the template for the form. In the `extrabody` block, we set a `translatable_file_uploader_options` variable that will deal with all translatable options for the file uploader, such as the widget template markup, error messages, and notifications:

```
{#templates/quotes/change_quote.html#}
{% extends "base.html" %}
{% load i18n crispy_forms_tags %}

{% block stylesheet %}
    {{ block.super }}
    <link rel="stylesheet" href="{% STATIC_URL %}ajaxuploader/css/
fileuploader.css" />
{% endblock %}

{% block content %}
    {% crispy form %}
{% endblock %}

{% block extrabody %}
    <script src="{% STATIC_URL %}ajaxuploader/js/fileuploader.
js"></script>
    <script>
        var translatable_file_uploader_options = {
            template: '<div class="qq-upload-drop-area"><span>{%
trans "Drop image here" %}</span></div>' +
```



```

        '<div class="qq-uploader">' +
        '<div class="qq-upload-button btn"><span
class="glyphicon glyphicon-upload"></span> {% trans "Upload
Image" %}</div>' +
        '&nbsp;<button class="btn btn-danger qq-delete-
button"><span class="glyphicon glyphicon-trash"></span> {% trans
"Delete" %}</button>' +
        '<ul class="qq-upload-list"></ul>' +
        '</div>',
        // template for one item in file list
        fileTemplate: '<li>' +
            '<span class="qq-upload-file"></span>' +
            '<span class="qq-upload-spinner"></span>' +
            '<span class="qq-upload-size"></span>' +
            '<a class="qq-upload-cancel" href="#">{% trans
"Cancel" %}</a>' +
            '<span class="qq-upload-failed-text">{% trans
"Failed" %}</span>' +
            '</li>',
        messages: {
            typeError: '{% trans "{file} has invalid
extension. Only {extensions} are allowed." %}',
            sizeError: '{% trans "{file} is too large, maximum
file size is {sizeLimit}." %}',
            minSizeError: '{% trans "{file} is too small,
minimum file size is {minSizeLimit}." %}',
            emptyError: '{% trans "{file} is empty, please
select files again without it." %}',
            filesLimitError: '{% trans "No more than
{filesLimit} files are allowed to be uploaded." %}',
            onLeave: '{% trans "The files are being uploaded,
if you leave now the upload will be cancelled." %}'
        }
    };
    var ajax_uploader_path = '{% url "ajax_uploader" %}';
</script>
<script src="{{ STATIC_URL }}site/js/change_quote.js"></
script>
{% endblock %}

```

6. Finally, we create the JavaScript file that will initialize the file upload widget and handle the image preview and deletion, as follows:

```
// site_static/site/js/change_quote.js
$(function() {
    var csrfmiddlewaretoken = $('input[name="csrfmiddlewaretoken"]').val();
    var $image_upload_widget = $('#image_upload_widget');
    var current_image_path = $('#id_picture_path').val();
    if (current_image_path) {
        $('.preview', $image_upload_widget).html(
            ''
        );
    }
    var options = $.extend(window.translatable_file_uploader_options, {
        allowedExtensions: ['jpg', 'jpeg', 'gif', 'png'],
        action: window.ajax_uploader_path,
        element: $('.uploader', $image_upload_widget)[0],
        multiple: false,
        onComplete: function(id, fileName, responseJSON) {
            if(responseJSON.success) {
                $('.messages', $image_upload_widget).html("");
                // set the original to media_file_path
                $('#id_picture_path').val('uploads/' + fileName);
                // show preview link
                $('.preview', $image_upload_widget).html(
                    ''
                );
            }
        },
        onAllComplete: function(uploads) {
            // uploads is an array of maps
            // the maps look like this: {file: FileObject,
            response: JSONServerResponse}
            $('.qq-upload-success').fadeOut("slow", function() {
                $(this).remove();
            });
        },
        params: {
            'csrf_token': csrfmiddlewaretoken,
            'csrf_name': 'csrfmiddlewaretoken',
            'csrf_xname': 'X-CSRFToken'
        }
    });
});
```

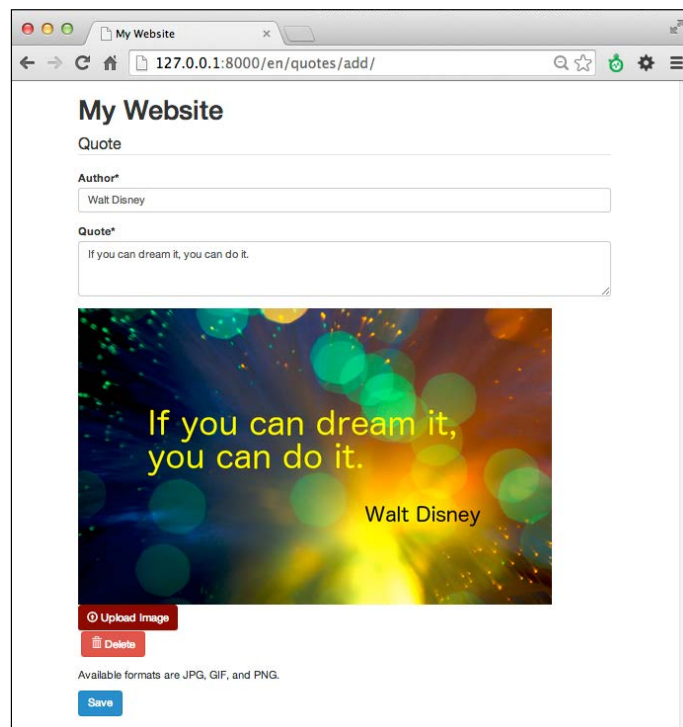
```

    },
    showMessage: function(message) {
        $('.messages', $image_upload_widget).html(
            '<div class="alert alert-danger">' + message + '</
div>'
        );
    }
});
var uploader = new qq.FileUploader(options);
$('.qq-delete-button', $image_upload_widget).click(function()
{
    $('.messages', $image_upload_widget).html("");
    $('.preview', $image_upload_widget).html("");
    $('#id_delete_picture').val(1);
    return false;
});
});

```

How it works...

When an image is selected in the upload widget, the result in the browser will look like this:



The same form can be used to create an inspirational quote and change an existing inspirational quote. Let's dig deeper into the process to see how it works. In the form, we have an uploading mechanism that consists of the following essential parts:

- ▶ The area for the preview of the image defined as a `<div>` tag with the "preview" CSS class. Initially, it might show an image if we are in an object change view, and the `InspirationQuote` object is passed to the template as `{{ instance }}`.
- ▶ The area for the Ajax uploader widget, defined as a `<div>` tag with the "uploader" CSS class. It will be filled in with dynamically created uploading and deleting buttons as well as with the uploading progress indicators.
- ▶ The help text for the upload.
- ▶ The area for error messages defined as a `<div>` tag with the "messages" CSS class.
- ▶ The hidden character field, `picture_path`, to set the path of the uploaded file.
- ▶ The hidden Boolean field, `delete_picture`, to mark the deletion of the file.

On page load, JavaScript will check if `picture_path` is set, and if it is, it will show a picture preview. This will be the case only when the form is submitted with an image selected, but there are validation errors.

Furthermore, there are options defined for the upload widget in JavaScript. These options are combined with the global `translatable-options` variable, `translatable_file_uploader_options`, set in the template, and functionality-specific options set in the JavaScript file. The Ajax upload widget is initialized with these options. Some important settings to note are the `onComplete` callback that shows an image preview and fills in the `picture_path` field when an image is uploaded, and the `showMessage` callback that defines how to show error messages in the wanted area.

Lastly, there is a handler for the delete button in JavaScript, which when clicked, sets 1 to the hidden field, `delete_picture`, and removes the preview image.

The Ajax uploader widget dynamically creates a form with the file upload field and a hidden `<iframe>` to post the form data. When a file is selected, it is immediately uploaded to the `uploads` directory under `MEDIA_URL`, and the path to the file is set to the hidden field, `picture_path`. This directory is a temporary location for the uploaded files. When the user submits the inspirational quote form, and the input is valid, the `save` method is called. If `delete_picture` is set to 1 and you are in the change form of the quote, the picture of the model instance will be deleted. If the `picture_path` field is defined, the image from the temporary location will be copied to its final destination and the original will be removed.

See also

- ▶ The *Uploading images* recipe in *Chapter 3, Forms and Views*
- ▶ The *Opening object details in a pop up* recipe
- ▶ The *Implementing a continuous scroll* recipe
- ▶ The *Implementing the Like widget* recipe

5

Custom Template Filters and Tags

In this chapter, we will cover the following recipes:

- ▶ Following conventions for your own template filters and tags
- ▶ Creating a template filter to show how many days have passed
- ▶ Creating a template filter to extract the first media object
- ▶ Creating a template filter to humanize URLs
- ▶ Creating a template tag to include a template if it exists
- ▶ Creating a template tag to load a QuerySet in a template
- ▶ Creating a template tag to parse content as a template
- ▶ Creating a template tag to modify request query parameters

Introduction

As you know, Django has quite an extensive template system, with features such as template inheritance, filters for changing the representation of values, and tags for presentational logic. Moreover, Django allows you to add your own template filters and tags in your apps. Custom filters or tags should be located in a template-tag library file under the `templatetags` Python package in your app. Your template-tag library can then be loaded in any template with a `{% load %}` template tag. In this chapter, we will create several useful filters and tags that give more control to the template editors.

Following conventions for your own template filters and tags

Custom template filters and tags can become a total mess if you don't have persistent guidelines to follow. Template filters and tags should serve template editors as much as possible. They should be both handy and flexible. In this recipe, we will look at some conventions that should be used when enhancing the functionality of the Django template system.

How to do it...

Follow these conventions when extending the Django template system:

1. Don't create or use custom template filters or tags when the logic for the page fits better in the view, context processors, or in model methods. When your page is context-specific, such as a list of objects or an object-detail view, load the object in the view. If you need to show some content on every page, create a context processor. Use custom methods of the model instead of template filters when you need to get some properties of an object not related to the context of the template.
2. Name the template-tag library with the `_tags` suffix. When your app is named differently than your template-tag library, you can avoid ambiguous package importing problems.
3. In the newly created library, separate filters from tags, for example, by using comments such as the following:

```
# -*- coding: UTF-8 -*-
from django import template
register = template.Library()

### FILTERS ###
# .. your filters go here..

### TAGS ###
# .. your tags go here..
```

4. Create template tags that are easy to remember by including the following constructs:
 - ❑ `for [app_name.model_name]`: Include this construct to use a specific model
 - ❑ `using [template_name]`: Include this construct to use a template for the output of the template tag
 - ❑ `limit [count]`: Include this construct to limit the results to a specific amount
 - ❑ `as [context_variable]`: Include this construct to save the results to a context variable that can be reused many times later

5. Try to avoid multiple values defined positionally in template tags unless they are self-explanatory. Otherwise, this will likely confuse the template developers.
6. Make as many arguments resolvable as possible. Strings without quotes should be treated as context variables that need to be resolved or as short words that remind you of the structure of the template tag components.

See also

- ▶ The *Creating a template filter to show how many days have passed* recipe
- ▶ The *Creating a template filter to extract the first media object* recipe
- ▶ The *Creating a template filter to humanize URLs* recipe
- ▶ The *Creating a template tag to include a template if it exists* recipe
- ▶ The *Creating a template tag to load a QuerySet in a template* recipe
- ▶ The *Creating a template tag to parse content as a template* recipe
- ▶ The *Creating a template tag to modify request query parameters* recipe

Creating a template filter to show how many days have passed

Not all people keep track of the date, and when talking about creation or modification dates of cutting-edge information, for many of us, it is more convenient to read the time difference, for example, the blog entry was posted three days ago, the news article was published today, and the user last logged in yesterday. In this recipe, we will create a template filter named `days_since` that converts dates to humanized time differences.

Getting ready

Create the `utils` app and put it under `INSTALLED_APPS` in the settings, if you haven't done that yet. Then, create a Python package named `templatetags` inside this app (Python packages are directories with an empty `__init__.py` file).

How to do it...

Create a `utility_tags.py` file with this content:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from datetime import datetime
```



```
from django import template
from django.utils.translation import ugettext_lazy as _
from django.utils.timezone import now as tz_now
register = template.Library()

### FILTERS ###

@register.filter
def days_since(value):
    """ Returns number of days between today and value."""

    today = tz_now().date()
    if isinstance(value, datetime.datetime):
        value = value.date()
    diff = today - value
    if diff.days > 1:
        return _("%s days ago") % diff.days
    elif diff.days == 1:
        return _("yesterday")
    elif diff.days == 0:
        return _("today")
    else:
        # Date is in the future; return formatted date.
        return value.strftime("%B %d, %Y")
```

How it works...

If you use this filter in a template like the following, it will render something like yesterday or 5 days ago:

```
{% load utility_tags %}
{{ object.created|days_since }}
```

You can apply this filter to the values of the date and datetime types.

Each template-tag library has a register where filters and tags are collected. Django filters are functions registered by the `register.filter` decorator. By default, the filter in the template system will be named the same as the function or the other callable object. If you want, you can set a different name for the filter by passing name to the decorator, as follows:

```
@register.filter(name="humanized_days_since")
def days_since(value):
    ...
```

The filter itself is quite self-explanatory. At first, the current date is read. If the given value of the filter is of the `datetime` type, the date is extracted. Then, the difference between today and the extracted value is calculated. Depending on the number of days, different string results are returned.

There's more...

This filter is easy to extend to also show the difference in time, such as `just now`, `7 minutes ago`, or `3 hours ago`. Just operate the `datetime` values instead of the `date` values.

See also

- The *Creating a template filter to extract the first media object* recipe
- The *Creating a template filter to humanize URLs* recipe

Creating a template filter to extract the first media object

Imagine that you are developing a blog overview page, and for each post, you want to show images, music, or videos in that page taken from the content. In such a case, you need to extract the ``, `<object>`, and `<embed>` tags out of the HTML content of the post. In this recipe, we will see how to do this using regular expressions in the `get_first_media` filter.

Getting ready

We will start with the `utils` app that should be set in `INSTALLED_APPS` in the settings and the `templatetags` package inside this app.

How to do it...

In the `utility_tags.py` file, add the following content:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
import re
from django import template
from django.utils.safestring import mark_safe
register = template.Library()

### FILTERS ###
```

```
media_file_regex = re.compile(r"<object .+?</object>|"
                               r"<(img|embed) [^>]+>")

@register.filter
def get_first_media(content):
    """ Returns the first image or flash file from the html
        content """
    m = media_file_regex.search(content)
    media_tag = ""
    if m:
        media_tag = m.group()
    return mark_safe(media_tag)
```

How it works...

While the HTML content in the database is valid, when you put the following code in the template, it will retrieve the `<object>`, ``, or `<embed>` tags from the content field of the object, or an empty string if no media is found there:

```
{% load utility_tags %}
{{ object.content|get_first_media }}
```

At first, we define the compiled regular expression as `media_file_regex`, then in the filter, we perform a search for that regular expression pattern. By default, the result will show the `<`, `>`, and `&` symbols escaped as `<`, `>`, and `&` entities. But we use the `mark_safe` function that marks the result as safe HTML ready to be shown in the template without escaping.

There's more...

It is very easy to extend this filter to also extract the `<iframe>` tags (which are more recently being used by Vimeo and YouTube for embedded videos) or the HTML5 `<audio>` and `<video>` tags. Just modify the regular expression like this:

```
media_file_regex = re.compile(r"<iframe .+?</iframe>|"
                               r"<audio .+?</ audio>|<video .+?</video>|"
                               r"<object .+?</object>|<(img|embed) [^>]+>")
```

See also

- ▶ [The *Creating a template filter to show how many days have passed* recipe](#)
- ▶ [The *Creating a template filter to humanize URLs* recipe](#)

Creating a template filter to humanize URLs

Usually, common web users enter URLs into address fields without protocol and trailing slashes. In this recipe, we will create a `humanize_url` filter used to present URLs to the user in a shorter format, truncating very long addresses, just like what Twitter does with the links in tweets.

Getting ready

As in the previous recipes, we will start with the `utils` app that should be set in `INSTALLED_APPS` in the settings, and should contain the `templatetags` package.

How to do it...

In the `FILTERS` section of the `utility_tags.py` template library in the `utils` app, let's add a filter named `humanize_url` and register it:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
import re
from django import template
register = template.Library()

### FILTERS ###

@register.filter
def humanize_url(url, letter_count):
    """ Returns a shortened human-readable URL """
    letter_count = int(letter_count)
    re_start = re.compile(r"^https?://")
    re_end = re.compile(r"/$")
    url = re_end.sub("", re_start.sub("", url))
    if len(url) > letter_count:
        url = u"%s..." % url[:letter_count - 1]
    return url
```

How it works...

We can use the `humanize_url` filter in any template like this:

```
{% load utility_tags %}
<a href="{{ object.website }}" target="_blank">
    {{ object.website|humanize_url:30 }}
</a>
```

The filter uses regular expressions to remove the leading protocol and the trailing slash, and then shortens the URL to the given amount of letters, adding an ellipsis to the end if the URL doesn't fit into the specified letter count.

See also

- ▶ [The Creating a template filter to show how many days have passed recipe](#)
- ▶ [The Creating a template filter to extract the first media object recipe](#)
- ▶ [The Creating a template tag to include a template if it exists recipe](#)

Creating a template tag to include a template if it exists

Django has the `{% include %}` template tag that renders and includes another template. However, in some particular situations, there is a problem that an error is raised if the template does not exist. In this recipe, we will show you how to create a `{% try_to_include %}` template tag that includes another template, but fails silently if there is no such template.

Getting ready

We will start again with the `utils` app that should be installed and is ready for custom template tags.

How to do it...

Template tags consist of two things: the function parsing the arguments of the template tag and the node class that is responsible for the logic of the template tag as well as for the output. Perform the following steps:

1. First, let's create the function parsing the template-tag arguments:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from django import template
from django.template.loader import get_template
register = template.Library()

### TAGS ###

@register.tag
def try_to_include(parser, token):
    """Usage: {% try_to_include "sometemplate.html" %}
```

```

This will fail silently if the template doesn't exist.
If it does, it will be rendered with the current context."""
try:
    tag_name, template_name = token.split_contents()
except ValueError:
    raise template.TemplateSyntaxError, \
        "%r tag requires a single argument" % \
            token.contents.split()[0]
return IncludeNode(template_name)

```

2. Then, we need the node class in the same file, as follows:

```

class IncludeNode(template.Node):
    def __init__(self, template_name):
        self.template_name = template_name

    def render(self, context):
        try:
            # Loading the template and rendering it
            template_name = template.resolve_variable(
                self.template_name, context)
            included_template = get_template(template_name).\
                render(context)
        except template.TemplateDoesNotExist:
            included_template = ""
        return included_template

```

How it works...

The `{% try_to_include %}` template tag expects one argument, that is, `template_name`. So, in the `try_to_include` function, we are trying to assign the split contents of the token only to the `tag_name` variable (which is "try_to_include") and the `template_name` variable. If this doesn't work, the template syntax error is raised. The function returns the `IncludeNode` object, which gets the `template_name` field for later usage.

In the `render` method of `IncludeNode`, we resolve the `template_name` variable. If a context variable was passed to the template tag, then its value will be used here for `template_name`. If a quoted string was passed to the template tag, then the content within quotes will be used for `template_name`.

Lastly, we try to load the template and render it with the current template context. If that doesn't work, an empty string is returned.

There are at least two situations where we could use this template tag:

- ▶ When including a template whose path is defined in a model, as follows:

```
{% load utility_tags %}
{% try_to_include object.template_path %}
```
- ▶ When including a template whose path is defined with the `{% with %}` template tag somewhere high in the template context variable's scope. This is especially useful when you need to create custom layouts for plugins in the placeholder of a template in Django CMS:

```
#templates/cms/start_page.html
{% with editorial_content_template_path=
"cms/plugins/editorial_content/start_page.html" %}
    {% placeholder "main_content" %}
{% endwith %}

#templates/cms/plugins/editorial_content.html
{% load utility_tags %}

{% if editorial_content_template_path %}
    {% try_to_include editorial_content_template_path %}
{% else %}
    <div>
        <!-- Some default presentation of
            editorial content plugin -->
    </div>
{% endif %}
```

There's more...

You can use the `{% try_to_include %}` tag as well as the default `{% include %}` tag to include templates that extend other templates. This has a beneficial use for large-scale portals where you have different kinds of lists in which complex items share the same structure as widgets but have a different source of data.

For example, in the artist list template, you can include the artist item template as follows:

```
{% load utility_tags %}
{% for object in object_list %}
    {% try_to_include "artists/includes/artist_item.html" %}
{% endfor %}
```

This template will extend from the item base as follows:

```
{# templates/artists/includes/artist_item.html #}
{% extends "utils/includes/item_base.html" %}

{% block item_title %}
    {{ object.first_name }} {{ object.last_name }}
{% endblock %}
```

The item base defines the markup for any item and also includes a Like widget, as follows:

```
{# templates/utils/includes/item_base.html #}
{% load likes_tags %}

<h3>{% block item_title %}{% endblock %}</h3>
{% if request.user.is_authenticated %}
    {% like_widget for object %}
{% endif %}
```

See also

- ▶ The *Creating templates for Django CMS* recipe in *Chapter 7, Django CMS*
- ▶ The *Writing your own CMS plugin* recipe in *Chapter 7, Django CMS*
- ▶ The *Implementing a Like* recipe in *Chapter 4, Templates and JavaScript*
- ▶ The *Creating a template tag to load a QuerySet in a template* recipe
- ▶ The *Creating a template tag to parse content as a template* recipe
- ▶ The *Creating a template tag to modify request query parameters* recipe

Creating a template tag to load a QuerySet in a template

Most often, the content that should be shown in a web page will have to be defined in the view. If this is the content to show on every page, it is logical to create a context processor. Another situation is when you need to show additional content such as the latest news or a random quote on some specific pages, for example, the start page or the details page of an object. In this case, you can load the necessary content with the `{% get_objects %}` template tag, which we will implement in this recipe.

Getting ready

Once again, we will start with the `utils` app that should be installed and ready for custom template tags.

How to do it...

Template tags consist of function parsing arguments passed to the tag and a node class that renders the output of the tag or modifies the template context. Perform the following steps:

1. First, let's create the function parsing the template-tag arguments, as follows:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from django.db import models
from django import template
register = template.Library()

### TAGS ###

@register.tag
def get_objects(parser, token):
    """
        Gets a queryset of objects of the model specified by app
    and
    model names
    Usage:
        {% get_objects [<manager>.<method> from
        <app_name>.<model_name> [limit <amount>] as
        <var_name> %}
    Example:
        {% get_objects latest_published from people.Person
        limit 3 as people %}
        {% get_objects site_objects.all from news.Article
        limit 3 as articles %}
        {% get_objects site_objects.all from news.Article
        as articles %}
    """
    amount = None
    try:
        tag_name, manager_method, str_from, appmodel,
        str_limit,\
        amount, str_as, var_name = token.split_contents()
    except ValueError:
        try:
            tag_name, manager_method, str_from, appmodel, str_as,\
            var_name = token.split_contents()
        except ValueError:
            raise template.TemplateSyntaxError, \
                "get_objects tag requires a following syntax: "
            "% get_objects [<manager>.<method> from "\
            "<app_name>.<model_name>"\
```

```

        " [limit <amount>] as <var_name> %}"
    try:
        app_name, model_name = appmodel.split(".")
    except ValueError:
        raise template.TemplateSyntaxError, \
            "get_objects tag requires application name and "\
            "model name separated by a dot"
    model = models.get_model(app_name, model_name)
    return ObjectsNode(model, manager_method, amount, var_name)

```

2. Then, we create the node class in the same file, as follows:

```

class ObjectsNode(template.Node):
    def __init__(self, model, manager_method, amount, var_name):
        self.model = model
        self.manager_method = manager_method
        self.amount = amount
        self.var_name = var_name

    def render(self, context):
        if "." in self.manager_method:
            manager, method = self.manager_method.split(".")
        else:
            manager = "_default_manager"
            method = self.manager_method

        qs = getattr(
            getattr(self.model, manager),
            method,
            self.model._default_manager.none,
        )()
        if self.amount:
            amount = template.resolve_variable(self.amount,
                                                context)
            context[self.var_name] = qs[:amount]
        else:
            context[self.var_name] = qs
        return ""

```

How it works...

The `{% get_objects %}` template tag loads a `QuerySet` defined by the `manager` method from a specified app and model, limits the result to the specified amount, and saves the result to a context variable.

This is the simplest example of how to use the template tag that we have just created. It will load five news articles in any template using the following snippet:

```
{% load utility_tags %}
{% get_objects all from news.Article limit 5 as latest_articles %}
{% for article in latest_articles %}
    <a href="{{ article.get_url_path }}">{{ article.title }}</a>
{% endfor %}
```

This is using the `all` method of the default `objects` manager of the `Article` model, and will sort the articles by the `ordering` attribute defined in the `Meta` class.

A more advanced example would be required to create a custom manager with a custom method to query objects from the database. A manager is an interface that provides database query operations to models. Each model has at least one manager called `objects` by default. As an example, let's create the `Artist` model, which has a `draft` or `published` status, and a new manager, `custom_manager`, which allows you to select random published artists:

```
#artists/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

STATUS_CHOICES = (
    ('draft', _("Draft")),
    ('published', _("Published")),
)

class ArtistManager(models.Manager):
    def random_published(self):
        return self.filter(status="published").order_by('?')

class Artist(models.Model):
    # ...
    status = models.CharField(_("Status"), max_length=20,
                              choices=STATUS_CHOICES)
    custom_manager = ArtistManager()
```

To load a random published artist, you add the following snippet to any template:

```
{% load utility_tags %}
{% get_objects custom_manager.random_published from artists.Artist
limit 1 as random_artists %}
{% for artist in random_artists %}
    {{ artist.first_name }} {{ artist.last_name }}
{% endfor %}
```

Let's look at the code of the template tag. In the parsing function, there is one of two formats expected: with the limit and without it. The string is parsed, the model is recognized, and then the components of the template tag are passed to the `ObjectNode` class.

In the `render` method of the node class, we check the manager's name and its method's name. If this is not defined, `_default_manager` will be used, which is, in most cases, the same as `objects`. After that, we call the manager method and fall back to empty `QuerySet` if the method doesn't exist. If the limit is defined, we resolve the value of it and limit the `QuerySet`. Lastly, we save the `QuerySet` to the context variable.

See also

- ▶ The *Creating a template tag to include a template if it exists* recipe
- ▶ The *Creating a template tag to parse content as a template* recipe
- ▶ The *Creating a template tag to modify request query parameters* recipe

Creating a template tag to parse content as a template

In this recipe, we will create a template tag named `{% parse %}`, which allows you to put template snippets into the database. This is valuable when you want to provide different content for authenticated and non-authenticated users, when you want to include a personalized salutation, or when you don't want to hardcode media paths in the database.

Getting ready

No surprise, we will start with the `utils` app that should be installed and ready for custom template tags.

How to do it...

Template tags consist of two things: the function parsing the arguments of the template tag and the node class that is responsible for the logic of the template tag as well as for the output. Perform the following steps:

1. First, let's create the function parsing the template-tag arguments, as follows:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from django import template
register = template.Library()
```

```
### TAGS ###

@register.tag
def parse(parser, token):
    """
        Parses the value as a template and prints it or saves to a
        variable
    Usage:
        {% parse <template_value> [as <variable>] %}
    Examples:
        {% parse object.description %}
        {% parse header as header %}
        {% parse "{{ MEDIA_URL }}" as js_url %}
    """
    bits = token.split_contents()
    tag_name = bits.pop(0)
    try:
        template_value = bits.pop(0)
        var_name = None
        if len(bits) == 2:
            bits.pop(0) # remove the word "as"
            var_name = bits.pop(0)
    except ValueError:
        raise template.TemplateSyntaxError, \
            "parse tag requires a following syntax: "\
            "{% parse <template_value> [as <variable>] %}"

    return ParseNode(template_value, var_name)
```

2. Then, we create the node class in the same file, as follows:

```
class ParseNode(template.Node):
    def __init__(self, template_value, var_name):
        self.template_value = template_value
        self.var_name = var_name

    def render(self, context):
        template_value = template.resolve_variable(
            self.template_value, context)
        t = template.Template(template_value)
        context_vars = {}
        for d in list(context):
            for var, val in d.items():
                context_vars[var] = val
            result = t.render(template.RequestContext(
                context['request'], context_vars))
        if self.var_name:
```

```

        context[self.var_name] = result
    return ""
    return result

```

How it works...

The `{% parse %}` template tag allows you to parse a value as a template and to render it immediately or to save it as a context variable.

If we have an object with a `description` field, which can contain template variables or logic, then we can parse it and render it using the following code:

```

{% load utility_tags %}
{% parse object.description %}

```

It is also possible to define a value to parse using a quoted string like this:

```

{% load utility_tags %}
{% parse "{{ STATIC_URL }}" as img_path %}


```

Let's have a look at the code of the template tag. The parsing function checks the arguments of the template tag bit by bit. At first, we expect the name `parse`, then the template value, then optionally the word `as`, and lastly the context variable name. The template value and the variable name are passed to the `ParseNode` class. The `render` method of that class at first resolves the value of the template variable and creates a template object out of it. Then, it renders the template with all the context variables. If the variable name is defined, the result is saved to it; otherwise, the result is shown immediately.

See also

- ▶ The *Creating a template tag to include a template if it exists* recipe
- ▶ The *Creating a template tag to load a QuerySet in a template* recipe
- ▶ The *Creating a template tag to modify request query parameters* recipe

Creating a template tag to modify request query parameters

Django has a convenient and flexible system to create canonical, clean URLs just by adding regular expression rules in the URL configuration files. But there is a lack of built-in mechanisms to manage query parameters. Views such as search or filterable object lists need to accept query parameters to drill down through filtered results using another parameter or to go to another page. In this recipe, we will create a template tag named `{% append_to_query %}`, which lets you add, change, or remove parameters of the current query.

Getting ready

Once again, we start with the `utils` app that should be set in `INSTALLED_APPS` and should contain the `templatetags` package.

Also, make sure that you have the `request` context processor set for the `TEMPLATE_CONTEXT_PROCESSORS` setting, as follows:

```
#settings.py
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.tz",
    "django.contrib.messages.context_processors.messages",
    "django.core.context_processors.request",
)
```

How to do it...

For this template tag, we will be using the `simple_tag` decorator that parses the components and requires you to define just the rendering function, as follows:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
import urllib
from django import template
from django.utils.encoding import force_str
register = template.Library()

### TAGS ###

@register.simple_tag(takes_context=True)
def append_to_query(context, **kwargs):
    """ Renders a link with modified current query parameters """
    query_params = context['request'].GET.copy()
    for key, value in kwargs.items():
        query_params[key] = value
    query_string = u""
    if len(query_params):
        query_string += u"?%s" % urllib.urlencode([
            (key, force_str(value)) for (key, value) in
            query_params.iteritems() if value
```

```
    ]).replace('&', '&amp;')
    return query_string
```

How it works...

The `{% append_to_query %}` template tag reads the current query parameters from the `request.GET` dictionary-like `QueryDict` object to a new dictionary named `query_params`, and loops through the keyword parameters passed to the template tag updating the values. Then, the new query string is formed, all spaces and special characters are URL-encoded, and ampersands connecting query parameters are escaped. This new query string is returned to the template.

To read more about `QueryDict` objects, refer to the official Django documentation:

<https://docs.djangoproject.com/en/1.6/ref/request-response/#querydict-objects>

Let's have a look at an example of how the `{% append_to_query %}` template tag can be used. If the current URL is `http://127.0.0.1:8000/artists/?category=fine-art&page=1`, we can use the following template tag to render a link that goes to the next page:

```
{% load utility_tags %}
<a href="{% append_to_query page=2 %}">2</a>
```

The following is the output rendered, using the preceding template tag:

```
<a href="?category=fine-art&page=2">2</a>
```

Or we can use the following template tag to render a link that resets pagination and goes to another category:

```
{% load utility_tags i18n %}
<a href="{% append_to_query category='sculpture' page='' %}">{% trans
"Sculpture" %}</a>
```

The following is the output rendered, using the preceding template tag:

```
<a href="?category=sculpture">Sculpture</a>
```

See also

- ▶ The *Filtering object lists* recipe in *Chapter 3, Forms and Views*
- ▶ The *Creating a template tag to include a template if it exists* recipe
- ▶ The *Creating a template tag to load a QuerySet in a template* recipe
- ▶ The *Creating a template tag to parse content as a template* recipe

6

Model Administration

In this chapter, we will cover the following recipes:

- ▶ Customizing columns in the change list page
- ▶ Creating admin actions
- ▶ Developing change list filters
- ▶ Exchanging administration settings for external apps
- ▶ Inserting a map into a change form

Introduction

The Django framework comes with a built-in administration system for your models. With very little effort, you can set up filterable, searchable, and sortable lists for browsing your models, and configure forms for adding and editing data. In this chapter, we will go through advanced techniques to customize administration by developing some practical cases.

Customizing columns in the change list page

Change list views in the default Django administration system let you have an overview of all instances of specific models. By default, the model admin property, `list_display`, controls which fields to show in different columns. But additionally, you can have custom functions set there that return data from relations or display custom HTML. In this recipe, we will create a special function for the `list_display` property that shows an image in one of the columns of the list view. As a bonus, we will make one field editable directly in the list view by adding the `list_editable` setting.

Getting ready

To start with, make sure that `django.contrib.admin` is in `INSTALLED_APPS` in the settings, and `AdminSite` is hooked into the URL configuration. Then, create a new app named `products` and put it under `INSTALLED_APPS`. This app will have the `Product` and `ProductPhoto` models, where one product might have multiple photos. For this example, we will also be using `UrlMixin`, which was defined in the *Creating a model mixin with URL-related methods* recipe in *Chapter 2, Database Structure*.

Let's create the `Product` and `ProductPhoto` models in the `models.py` file as follows:

```
#products/models.py
# -*- coding: UTF-8 -*-
import os
from django.db import models
from django.utils.timezone import now as timezone_now
from django.utils.translation import ugettext_lazy as _
from django.core.urlresolvers import reverse
from django.core.urlresolvers import NoReverseMatch
from utils.models import UrlMixin

def upload_to(instance, filename):
    now = timezone_now()
    filename_base, filename_ext = os.path.splitext(filename)
    return "products/%s/%s%s" % (
        instance.product.slug,
        now.strftime("%Y%m%d%H%M%S"),
        filename_ext.lower(),
    )

class Product(UrlMixin):
    title = models.CharField(_("title"), max_length=200)
    slug = models.SlugField(_("slug"), max_length=200)
    description = models.TextField(_("description"), blank=True)
    price = models.DecimalField(_(u"price (€)"), max_digits=8,
                               decimal_places=2, blank=True, null=True)

    class Meta:
        verbose_name = _("Product")
        verbose_name_plural = _("Products")

    def __unicode__(self):
        return self.title
```

```

def get_url_path(self):
    try:
        return reverse("product_detail", kwargs={
            "slug": self.slug})
    except NoReverseMatch:
        return ""

class ProductPhoto(models.Model):
    product = models.ForeignKey(Product)
    photo = models.ImageField(_("photo"), upload_to=upload_to)

    class Meta:
        verbose_name = _("Photo")
        verbose_name_plural = _("Photos")

    def __unicode__(self):
        return self.photo.name

```

How to do it...

We will create a simple administration for the `Product` model that will have instances of the `ProductPhoto` model attached to the product as inlines.

In the `list_display` property, we will define the `get_photo` method name of the model admin that will be used to show the first photo from the many-to-one relationship.

Let's create an `admin.py` file with the following content:

```

#products/admin.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.contrib import admin
from django.utils.translation import ugettext_lazy as _
from django.http import HttpResponseRedirect

from products.models import Product, ProductPhoto

class ProductPhotoInline(admin.StackedInline):
    model = ProductPhoto
    extra = 0

class ProductAdmin(admin.ModelAdmin):
    list_display = ["title", "get_photo", "price"]
    list_editable = ["price"]

    fieldsets = (

```

```

        (_("Product"), {
            "fields": ("title", "slug", "description", "price"),
        }),
    )
    prepopulated_fields = {"slug": ("title",)}
    inlines = [ProductPhotoInline]





    def get_photo(self, obj):
        project_photos = obj.productphoto_set.all()[:1]
        if project_photos.count() > 0:
            return u"" <a href="%s" target="_blank">
                
            </a>"" % {
                "product_url": obj.get_url_path(),
                "photo_url": project_photos[0].photo.url,
            }
        return u""
    get_photo.short_description = _("First photo")
    get_photo.allow_tags = True

admin.site.register(Product, ProductAdmin)

```

How it works...

If you look at the product administration list in the browser, it will look like this:

<input type="checkbox"/> Title	First photo	Price (€)
<input type="checkbox"/> Detroit Electric Car		<input type="text"/>
<input type="checkbox"/> Loopwheels on Dahon Mu		<input type="text" value="1768"/>
<input type="checkbox"/> Mercury Skate		<input type="text"/>
<input type="checkbox"/> Ryno		<input type="text" value="3865"/>

4 Products Save

Besides the normal field names, the `list_display` property accepts a function or another callable, the name of an attribute of the admin model, or the name of the attribute of the model. Each callable will be passed a model instance as the first argument. So, in our example, we have the `get_photo` method of the model `admin` that retrieves `Product` as `obj`. The method tries to get the first `ProductPhoto` from the many-to-one relation and, if it exists, it returns HTML with the `` tag linked to the detail page of `Product`.

The `short_description` property of the callable defines the title shown for the column. The `allow_tags` property tells the administration not to escape the HTML values.

In addition, the price field is made editable by the `list_editable` setting and there is a save button at the bottom to save the whole list of products.

See also

- ▶ The *Creating a model mixin with URL-related methods* recipe in *Chapter 2, Database Structure*
- ▶ The *Creating admin actions* recipe
- ▶ The *Developing change list filters* recipe

Creating admin actions

The Django administration system provides actions that we can execute for selected items in the list. There is one action given by default, and it is used to delete selected instances. In this recipe, we will create an additional action for the list of the `Product` model that allows administrators to export selected products to Excel spreadsheets.

Getting ready

We will start with the `products` app that we created in the previous recipe.

How to do it...

Admin actions are functions that take three arguments: the current `ModelAdmin` value, the current `HttpRequest` value, and the `QuerySet` value containing the selected items. Perform the following steps:

1. Let's create an `export_xls` function in the `admin.py` file of the `products` app, as follows:

```
#products/admin.py
# -*- coding: UTF-8 -*-
import xlwt
```

```
# ... other imports ...

def export_xls(modeladmin, request, queryset):
    response = HttpResponse(mimetype="application/ms-excel")
    response["Content-Disposition"] = "attachment; "\
        "filename=products.xls"
    wb = xlwt.Workbook(encoding="utf-8")
    ws = wb.add_sheet("Products")

    row_num = 0

    ### Print Title Row ###    columns = [
        # column name, column width
        (u"ID", 2000),
        (u"Title", 6000),
        (u"Description", 8000),
        (u"Price (€)", 3000),
    ]

    header_style = xlwt.XFStyle()
    header_style.font.bold = True

    for col_num in xrange(len(columns)):
        ws.write(row_num, col_num, columns[col_num][0],
            header_style)
        # set column width
        ws.col(col_num).width = columns[col_num][1]

    ### Print Content ###

    text_style = xlwt.XFStyle()
    text_style.alignment.wrap = 1

    price_style = xlwt.XFStyle()
    price_style.num_format_str = "0.00"

    styles = [text_style, text_style, text_style, price_style]

    for obj in queryset.order_by("pk"):
        row_num += 1
        row = [
            obj.pk,
            obj.title,
```

```

        obj.description,
        obj.price,
    ]
    for col_num in xrange(len(row)):
        ws.write(row_num, col_num, row[col_num],
                styles[col_num])

    wb.save(response)
    return response

export_xls.short_description = u"Export XLS"

```

- Then, add the actions setting to ProductAdmin, as follows:

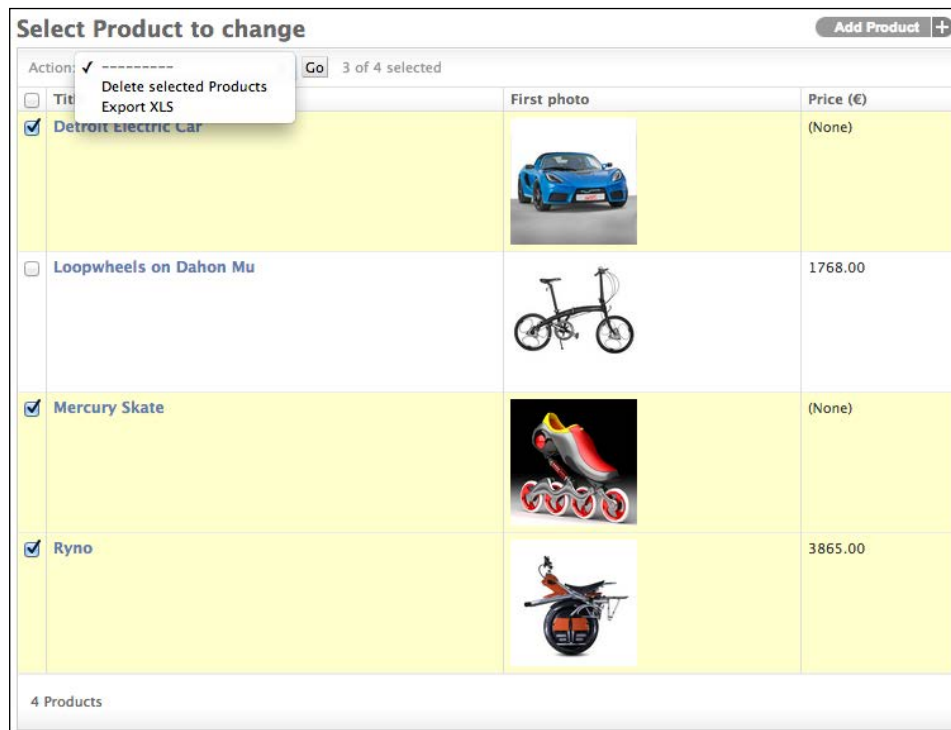
```

class ProductAdmin(admin.ModelAdmin):
    # ...
    actions = [export_xls]

```

How it works...

If you look at the product administration list page in the browser, you will see a new action called **Export XLS** along with the default action **Delete selected Products**:



By default, admin actions do something with `QuerySet` and redirect the administrator back to the change list page. However, for some more complex actions like this, `HttpResponse` can be returned. The `export_xls` function returns `HttpResponse` with the MIME type of Excel spreadsheet. Using the `Content-Disposition` header, we set the response to be downloadable with the file named `products.xls`.

Then, we use the `xlwt` Python module to create the Excel file.

At first, the workbook with UTF-8 encoding is created. Then, we add a sheet named `Products` to it. We will be using the `write` method of the sheet to set the content and style for each cell and the `col` method to retrieve the column and set the width to it.

To have an overview of all columns in the sheet, we create a list of tuples with column names and widths. Excel uses some magical units for the widths of the columns. They are 1/256 of the width of the zero character for the default font. Next, we define the header style to be bold. As we have the columns defined, we loop through them and fill the first row with the column names, also assigning the bold style to them.

Then, we create a style for normal cells and for the prices. The text in normal cells will be wrapped in multiple lines. Prices will have a special number style with two points after the decimal point.

Lastly, we go through the `QuerySet` of the selected products ordered by ID and print the specified fields into corresponding cells, also applying specific styles.

The workbook is saved to the file-like `HttpResponse` object and the resulting Excel sheet looks like this:

ID	Title	Description	Price (€)
1	Ryno	With the Ryno microcycle, you're not limited to the street or the bike lane. It's a transitional vehicle—it goes most places where a person can walk or ride a bike.	3865.00
2	Mercury Skate	The main purpose of designing this Mercury Skate is to decrease the skater's fatigue and provide them with an easier and smoother ride on the pavement.	
4	Detroit Electric Car	The Detroit Electric SP:01 is a limited-edition, two-seat, pure-electric sports car that sets new standards for performance and handling in electric vehicles.	

See also

- ▶ *Chapter 9, Data Import and Export*
- ▶ *The Customizing columns in the change list page recipe*
- ▶ *The Developing change list filters recipe*

Developing change list filters

If you want administrators to be able to filter the change list by date, relation, or field choices, you need to use the `list_filter` property for the admin model. In addition, there is a possibility to have custom-tailored filters. In this recipe, we will add a filter that allows you to select products by the number of photos attached to them.

Getting ready

Let's start with the `products` app that we created in the previous recipes.

How to do it...

Execute these two steps:

1. In the `admin.py` file, create a `PhotoFilter` class extending from `SimpleListFilter`:

```
#products/admin.py
# -*- coding: UTF-8 -*-
# ... all previous imports go here ...
from django.db import models

class PhotoFilter(admin.SimpleListFilter):
    # Human-readable title which will be displayed in the
    # right admin sidebar just above the filter options.
    title = _("photos")

    # Parameter for the filter that will be used in the URL query.
    parameter_name = "photos"

    def lookups(self, request, model_admin):
        """
        Returns a list of tuples. The first element in each
        tuple is the coded value for the option that will
        appear in the URL query. The second element is the
        human-readable name for the option that will appear
```

```
in the right sidebar.
"""
return (
    ("0", _("Has no photos")),
    ("1", _("Has one photo")),
    ("2+", _("Has more than one photo")),
)

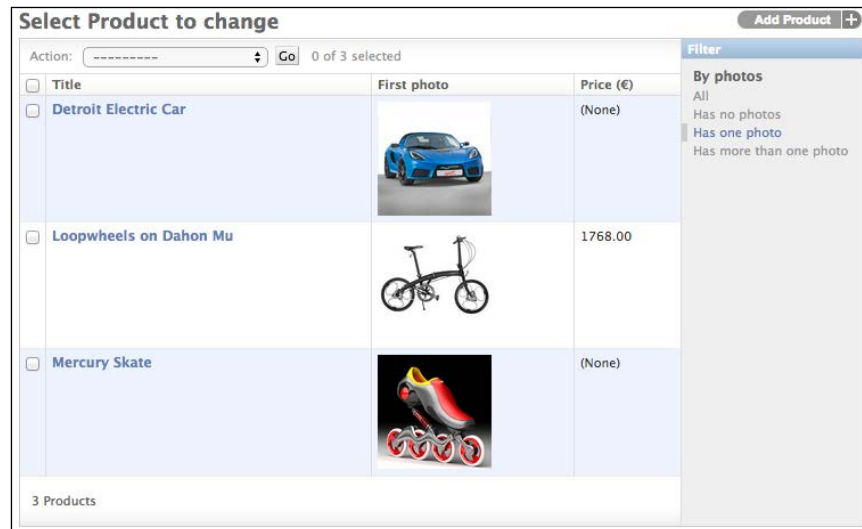
def queryset(self, request, queryset):
    """
    Returns the filtered queryset based on the value
    provided in the query string and retrievable via
    `self.value()`.
    """
    if self.value() == "0":
        return queryset.annotate(
            num_photos=models.Count("productphoto")
        ).filter(num_photos=0)
    if self.value() == "1":
        return queryset.annotate(
            num_photos=models.Count("productphoto")
        ).filter(num_photos=1)
    if self.value() == "2+":
        return queryset.annotate(
            num_photos=models.Count("productphoto")
        ).filter(num_photos__gte=2)
```

2. Then, add a list filter to ProductAdmin:

```
class ProductAdmin(admin.ModelAdmin):
    # ...
    list_filter = [PhotoFilter]
```

How it works...

The list filter that we just created will be shown in the sidebar of the product list, as follows:



The `PhotoFilter` class has translatable title and query parameter names as properties as well as two methods: `lookups`, defining the choices of the filter, and `queryset`, defining how to filter the `QuerySet` of objects when a specific value is selected.

In the `lookups` method, we define three choices: there are no photos, there is one photo, and there is more than one photo attached. In the `queryset` method, we use the `annotate` method of the `QuerySet` to select the count of photos for each product. This count of the photos is then filtered according to the selected choice.

To learn more about the aggregation functions such as `annotate`, refer to the official Django documentation:

<https://docs.djangoproject.com/en/1.6/topics/db/aggregation/>

See also

- ▶ The *Customizing columns in the change list page* recipe
- ▶ The *Creating admin actions* recipe
- ▶ The *Exchanging administration settings for external apps* recipe

Exchanging administration settings for external apps

Django apps as well as third-party apps come with their own administration settings, but there is a mechanism to switch these settings off and use your own better administration settings. In this recipe, you will learn how to exchange the administration settings for the `django.contrib.auth` app with custom administration settings.

Getting ready

Create an app named `custom_admin` with an empty `models.py` file and put this app under `INSTALLED_APPS` in the settings.

How to do it...

Insert the following content in the new file named `admin.py`, in the `custom_admin` app:

```
#custom_admin/admin.py
# -*- coding: UTF-8 -*-
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin, GroupAdmin
from django.contrib.auth.admin import User, Group
from django.utils.translation import ugettext_lazy as _

class UserAdminExtended(UserAdmin):
    list_display = ("username", "email", "first_name", "last_name",
                    "is_active", "is_staff", "date_joined", "last_login")
    list_filter = ("is_active", "is_staff", "is_superuser",
                   "date_joined", "last_login")
    ordering = ("last_name", "first_name", "username")
    save_on_top = True

class GroupAdminExtended(GroupAdmin):
    list_display = ("__unicode__", "display_users")
    save_on_top = True

    def display_users(self, obj):
        links = []
        for user in obj.user_set.all():
            links.append(
                """<a href="/admin/auth/user/%d/" target="_blank">%s
                </a>""" % (
                    user.id,
```

```

        (u"%s %s" % (user.first_name, user.last_
name)).\
        strip() or user.username,
    )
    )
    return u"<br />".join(links)
display_users.allow_tags = True
display_users.short_description = _("Users")

admin.site.unregister(User)
admin.site.unregister(Group)
admin.site.register(User, UserAdminExtended)
admin.site.register(Group, GroupAdminExtended)

```

How it works...

Here we created two model admin classes, `UserAdminExtended` and `GroupAdminExtended`, which extend from the contributed `UserAdmin` and `GroupAdmin` classes respectively and overwrite some of the properties. Then, we unregistered the existing administration classes for the `User` and `Group` models and registered some new modified ones.

The modified user administration settings show more fields than the default settings in the list view, and also add additional filters and ordering options and show submit buttons at the top of the editing form.

In the change list of the new group administration settings, we display users who are assigned to specific groups. That looks like this in the browser:

Group	Users
<input type="checkbox"/> Editors	Aidas Bendoraitis Erika Mustermann Jean Dupont John Doe Vardenis Pavardenis

1 group

See also

- ▶ The *Customizing columns in the change list page* recipe
- ▶ The *Inserting a map into a change form* recipe

Inserting a map into a change form

In this recipe, we will create the `locations` app with the `Location` model and overwrite the template of the change form to add a Google map where an administrator can find and mark geographical coordinates of the location.

Getting ready

We will start with the `locations` app that should be put under `INSTALLED_APPS` in the settings. Create a `Location` model there with the title, description, address, and geographical coordinates, as follows:

```
#locations/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

COUNTRY_CHOICES = (
    ("UK", _("United Kingdom")),
    ("DE", _("Germany")),
    ("FR", _("France")),
    ("LT", _("Lithuania")),
)

class Location(models.Model):
    title = models.CharField(_("title"), max_length=255, unique=True)
    description = models.TextField(_("description"), blank=True)
    street_address = models.CharField(_("street address"),
                                      max_length=255, blank=True)
    street_address2 = models.CharField(_("street address (2nd line)"),
                                       max_length=255, blank=True)
    postal_code = models.CharField(_("postal code"), max_length=10,
                                   blank=True)
    city = models.CharField(_("city"), max_length=255, blank=True)
    country = models.CharField(_("country"), max_length=2, blank=True,
                              choices=COUNTRY_CHOICES)
    latitude = models.FloatField(_("latitude"), blank=True, null=True,
                                 help_text=_("Latitude (Lat.) is the angle between any point "
                                             "and the equator "
                                             "(north pole is at 90; south pole is at -90)."))
    longitude = models.FloatField(_("longitude"), blank=True,
                                  null=True,
```

```

        help_text=_("Longitude (Long.) is the angle east or west of "
            "an arbitrary point on Earth from Greenwich (UK), "
            "which is the international zero-longitude point "
            "(longitude=0 degrees). "
            "The anti-meridian of Greenwich is both 180 "
            "(direction to east) and -180 (direction to west).")
    class Meta:
        verbose_name = _("Location")
        verbose_name_plural = _("Locations")

    def __unicode__(self):
        return self.title

```

How to do it...

The administration of the Location model is as simple as it can be. Perform the following steps:

1. Note that we are grouping the fields into fieldsets, as follows:

```

#locations/admin.py
# -*- coding: UTF-8 -*-
from django.utils.translation import ugettext_lazy as _
from django.contrib import admin
from models import Location

class LocationAdmin(admin.ModelAdmin):
    save_on_top = True
    list_display = ("title", "street_address", "description")
    search_fields = ("title", "street_address", "description")

    fieldsets = [
        (_("Main Data"), {"fields": ("title", "description")}),
        (_("Address"), {"fields": ("street_address",
            "street_address2", "postal_code", "city",
            "country", "latitude", "longitude")}),
    ]

admin.site.register(Location, LocationAdmin)

```

2. To create a custom change form template, add a new file called `change_form.html` under `admin/locations/location/` in your templates directory. This template will extend from the default `admin/change_form.html` template and will overwrite the `extrastyle` and `field_sets` blocks, as follows:

```

{#templates/admin/locations/location/change_form.html#}
{% extends "admin/change_form.html" %}

```



```
{% load i18n admin_static admin_modify %}
{% load url from future %}
{% load admin_urls %}

{% block extrastyle %}
    {{ block.super }}
    <style type="text/css">
        #map_canvas {
            width:722px;
            height:300px;
        }
        .buttonHolder {
            width:722px;
        }
        #remove_geo {
            float: right;
        }
    </style>
{% endblock %}

{% block field_sets %}
    {% for fieldset in adminform %}
        {% include "admin/includes/fieldset.html" %}
        {% if forloop.counter == 2 %}
            <fieldset class="module aligned">
                <h2>{% trans "Map" %}</h2>
                <div class="form-row">
                    <div id="map_canvas">
                        <!-- THE GMAPS WILL BE INSERTED HERE
DYNAMICALLY -->
                    </div>
                    <ul id="map_locations"></ul>
                    <div class="buttonHolder">
                        <button id="locate_address" type="button"
class="secondaryAction">
                            {% trans "Locate address" %}
                        </button>&nbsp;
                        <button id="remove_geo" type="button"
class="secondaryAction">
                            {% trans "Remove from map" %}
                        </button>
                    </div>
                </div>
            </div>
        {% endif %}
    {% endfor %}
{% endblock %}
```

```

        </fieldset>
        {% endif %}
    {% endfor %}

    <script type="text/javascript" src="//ajax.googleapis.com/
ajax/libs/jquery/1.9.1/jquery.min.js"></script>
    <script type="text/javascript" src="http://maps.google.com/
maps/api/js?sensor=false&language=en&region=UK"></script>
    <script type="text/javascript" src="{ { STATIC_URL } }site/js/
locating.js"></script>
    {% endblock %}

```

3. Then, let's create a JavaScript file named `locating.js`. We will be using jQuery in this file, as jQuery comes in the contributed administration system and makes the work easy and cross-browser. We don't want to pollute the environment with global variables, so we will start with a closure to make a private scope for variables and functions (a closure is the local variables for a function kept alive after the function has returned):

```

//site_static/site/js/locating.js
(function ($, undefined) {
    var gMap;
    var gettext = self.gettext || function (val) {return val;};
    var gMarker;

    // ... this is where all the functions go ...

})(jQuery);

```

4. Then, we will create JavaScript functions one by one.
5. The `getAddress4search` function will collect the address string from address fields that can later be used for geocoding, as follows:

```

function getAddress4search() {
    var address = [];
    var sStreetAddress2 = $('#id_street_address2').val();
    if (sStreetAddress2) {
        sStreetAddress2 = ' ' + sStreetAddress2;
    }
    address.push($('#id_street_address').val() + sStreetAddress2);
    address.push($('#id_city').val());
    address.push($('#id_country').val());
    address.push($('#id_postal_code').val());
    return address.join(', ');
}

```

6. The `updateMarker` function will take the `latitude` and `longitude` arguments and will draw or move a marker on the map. Also, it makes the marker draggable:

```
function updateMarker(lat, lng) {
    var point = new google.maps.LatLng(lat, lng);
    if (gMarker) {
        gMarker.setPosition(point);
    } else {
        gMarker = new google.maps.Marker({
            position: point,
            map: gMap
        });
    }
    gMap.panTo(point, 15);
    gMarker.setDraggable(true);
    google.maps.event.addListener(gMarker, 'dragend', function() {
        var point = gMarker.getPosition();
        updateLatitudeAndLongitude(point.lat(), point.lng());
    });
}
```

7. The `updateLatitudeAndLongitude` function takes the `latitude` and `longitude` arguments and updates the values for the fields with these IDs, `id_latitude` and `id_longitude`, as follows:

```
function updateLatitudeAndLongitude(lat, lng) {
    lat = Math.round(lat * 1000000) / 1000000;
    lng = Math.round(lng * 1000000) / 1000000;
    $('#id_latitude').val(lat);
    $('#id_longitude').val(lng);
}
```

8. The `autocompleteAddress` function gets the results from Google geocoding and lists them below the map to select the correct one, or if there is just one result, it updates the geographical position and address fields, as follows:

```
function autocompleteAddress(results) {
    var $foundLocations = $('#map_locations').html('');
    var i, len = results.length;

    if (results) {
        if (len > 1) {
            $(
                '<a href="">' + results[i].formatted_address + '</a>'
            ).data("gmap_index", i).click(function (e) {
                e.preventDefault();
                var result = results[$(this).data("gmap_index")];
                updateAddressFields(result.address_components);
                var point = result.geometry.location;
            });
        }
    }
}
```

```

        updateLatitudeAndLongitude(point.lat(), point.lng());
        updateMarker(point.lat(), point.lng());
        $foundLocations.hide();

        }).appendTo($('li').appendTo($foundLocations));
    }
    $('a href="" + gettext("None of the listed")
+ '</a>').click(function (e) {
        e.preventDefault();
        $foundLocations.hide();

        }).appendTo($('li').appendTo($foundLocations));
    $foundLocations.show();
} else {
    $foundLocations.hide();
    var result = results[0];
    updateAddressFields(result.address_components);
    var point = result.geometry.location;
    updateLatitudeAndLongitude(point.lat(), point.lng());
    updateMarker(point.lat(), point.lng());
}
}
}

```

9. The `updateAddressFields` function takes a nested dictionary with address components as an argument and fills in all address fields:

```

function updateAddressFields(addressComponents) {
    var i, len=addressComponents.length;
    var streetName, streetNumber;
    for (i=0; i<len; i++) {
        var obj = addressComponents[i];
        var obj_type = obj.types[0];
        if (obj_type == 'locality') {
            $('#id_city').val(obj.long_name);
        }
        if (obj_type == 'street_number') {
            streetNumber = obj.long_name;
        }
        if (obj_type == 'route') {
            streetName = obj.long_name;
        }
        if (obj_type == 'postal_code') {
            $('#id_postal_code').val(obj.long_name);
        }
        if (obj_type == 'country') {

```

```
        $('#id_country').val(obj.short_name);
    }
}
if (streetName) {
    var streetAddress = streetName;
    if (streetNumber) {
        streetAddress += ' ' + streetNumber;
    }
    $('#id_street_address').val(streetAddress);
}
}
```

10. Finally, we have the initialization function that is called on page load. It attaches the onclick event handlers to the buttons, creates a Google map, and initially marks the geoposition defined in the latitude and longitude fields, as follows:

```
$(document).ready(function () {
    $('#locate_address').click(function() {
        var oGeocoder = new google.maps.Geocoder();
        oGeocoder.geocode(
            {address: getAddress4search()},
            function (results, status) {
                if (status === google.maps.GeocoderStatus.OK) {
                    autocompleteAddress(results);
                } else {
                    autocompleteAddress(false);
                }
            }
        );
    });

    $('#remove_geo').click(function() {
        $('#id_latitude').val('');
        $('#id_longitude').val('');
        gMarker.setMap(null);
        gMarker = null;
    });

    gMap = new google.maps.Map($('#map_canvas').get(0), {
        scrollwheel: false,
        zoom: 16,
        center: new google.maps.LatLng(51.511214, -0.119824),
        disableDoubleClickZoom: true
    });
    google.maps.event.addListener(gMap, 'dblclick',
    function(event) {
        updateLatitudeAndLongitude(event.latLng.lat(), event.
```

```

latLng.lng());
    updateMarker(event.latLng.lat(), event.latLng.lng());
});
$('#map_locations').hide();

var $lat = $('#id_latitude');
var $lng = $('#id_longitude');
if ($lat.val() && $lng.val()) {
    updateMarker($lat.val(), $lng.val());
}
});

```

How it works...

If you look at the location change form in the browser, you will see the map shown in the fieldset with address fields:

The screenshot shows a web form with two main sections: 'Address' and 'Map'.

Address Section:

- Street address:
- Street address (2nd line):
- Postal code:
- City:
- Country:
- Latitude:
Latitude (Lat.) is the angle between any point and the equator (north pole is at 90; south pole is at -90).
- Longitude:
Longitude (Long.) is the angle east or west of an arbitrary point on Earth from Greenwich (UK), which is the international zero-longitude point (longitude=0 degrees). The anti-meridian of Greenwich is both 180 (direction to east) and -180 (direction to west).

Map Section:

The map shows a street view of Berlin, Germany, with a red pin marking the location of 'Haus der Kulturen der Welt' (HKW) on John-Foster-Dulles-Allee. The map includes labels for 'Spree', 'Museuminsel', 'German Chancellery', and 'Forum am Kanzleramt'. The map data is attributed to 2014 GeoBasis-DE/BKG (©2009), Google.

At the bottom of the form, there are buttons: 'Delete', 'Save and add another', 'Save and continue editing', and 'Save'.

Below the map, there are two buttons, **Locate address** and **Remove from map**.

When you click on the **Locate address** button, the geocoding is called to search for the geographical coordinates of the entered address. The result of geocoding is either one or more addresses with latitudes and longitudes in a nested dictionary format. To see the structure of the nested dictionary in the console of developer tools, just put this line at the beginning of the `autocompleteAddress` function:

```
console.log(JSON.stringify(results, null, 4));
```

If there is just one result, the missing postal code or other missing address fields are populated, the latitude and longitude are filled in, and a marker is put at a specific place on the map. If there are more results, the full list of them is shown below the map with the option to select the correct one.

Then, the administrator can move the marker on the map by dragging-and-dropping. Also, a double-click anywhere on the map updates the geographical coordinates and the marker position.

Finally, if the **Remove from map** button is clicked, the geographical coordinates are cleaned and the marker is removed.

See also

- The *Using HTML5 data attributes* recipe in *Chapter 4, Templates and JavaScript*

7

Django CMS

In this chapter, we will cover the following recipes:

- ▶ Creating templates for Django CMS
- ▶ Structuring the page menu
- ▶ Converting an app to a CMS app
- ▶ Attaching your own navigation
- ▶ Writing your own CMS plugin
- ▶ Adding new fields to the CMS page

Introduction

Django CMS is an open source content management system based on Django and created by Divio AG in Switzerland. Django CMS takes care of a website's structure, provides navigation menus, makes it easy to edit page content in the frontend, supports multiple languages in a website, and you can also extend it to your needs using the provided hooks. To create a website, you need to create a hierarchical structure of pages where each page has a template. Templates have placeholders that can be assigned different plugins with the content. Using special template tags, menus can be generated out of the hierarchical page structure. The CMS takes care of URL mapping to specific pages.

In this chapter, we will look into Django CMS 3.0 from a developer's perspective. We will see what is necessary for the templates to function and have a look at the possible page structure for header and footer navigation. You will also learn how to attach URL rules of an app to a CMS page. Then, we will attach custom navigation to the CMS page tree and create our own CMS content plugins. And finally, you will learn how to add new fields to the CMS pages.

Although in this book I won't guide you through all the bits and pieces of using Django CMS, by the end of the chapter, you should be aware of what is possible to do with it. The rest can be learned from the official documentation and especially by trying out the frontend user interface of the CMS.

Creating templates for Django CMS

For every page in your page structure, you need to choose a template from the list of templates defined in the settings. In this recipe, we will look at the minimum requirements for those templates.

Getting ready

To start, follow the official Django CMS tutorial to install Django CMS to your virtual environment:

<https://github.com/divio/django-cms-tutorial>

How to do it...

We will update the `base.html` template so that it contains everything that Django CMS needs, and then we will create and register two templates, `default.html` and `start.html`, to choose from for CMS pages:

1. First of all, we will update the base template that we created in the *Arranging the base.html template* recipe in *Chapter 4, Templates and JavaScript*, as follows:

```
{#templates/base.html#}
{% block doctype %}<!DOCTYPE html>{% endblock %}
{% load i18n cms_tags sekizai_tags menu_tags %}
<html lang="{% LANGUAGE_CODE %}">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1" />
    <title>{% block title %}{% endblock %}{% trans "My Website"
%}</title>
    <link rel="icon" href="{% STATIC_URL %}site/img/favicon.ico"
type="image/png" />

    {% block meta_tags %}{% endblock %}

    {% render_block "css" %}
    {% block stylesheet %}
```

```

        <link rel="stylesheet"
href="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/
bootstrap.min.css" />
        <link href="{ { STATIC_URL } }site/css/style.css"
rel="stylesheet" media="screen" type="text/css" />
        {% endblock %}

    <script
src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
    <script
src="http://code.jquery.com/jquery-migrate-1.2.1.min.js"></script>
    <script
src="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js/bootstrap.
min.js"></script>

    {% block js %}{% endblock %}
    {% block extrahead %}{% endblock %}
</head>
<body class="{% block bodyclass %}{% endblock %}">
    {% cms_toolbar %}
    {% block page %}
        <div class="wrapper">
            <div id="header" class="clearfix container">
                <h1>{% trans "My Website" %}</h1>
                <nav class="navbar navbar-default"
role="navigation">
                    {% block header_navigation %}
                        <ul class="nav navbar-nav">
                            {% show_menu_below_id "start_page" 0 1
1 1 %}

                        </ul>
                    {% endblock %}
                    {% block language_chooser %}
                        <ul class="nav navbar-nav pull-right">
                            {% language_chooser %}
                        </ul>
                    {% endblock %}
                </nav>
            </div>
            <div id="content" class="clearfix container">
                {% block content %}
                {% endblock %}
            </div>
            <div id="footer" class="clearfix container">
                {% block footer_navigation %}
                <nav class="navbar navbar-default"
role="navigation">

```

```
        <ul class="nav navbar-nav">
            {%
show_menu_below_id "footer_navigation" 0 1 1 1 %}
        </ul>
    </nav>
    {% endblock %}
</div>
</div>
{% endblock %}
{% block extrabody %}{% endblock %}
{% render_block "js" %}
</body>
</html>
```

2. Then, we will create a directory named `cms` under `templates` and add two templates for CMS pages, `default.html` for normal pages and `start.html` for the home page, as follows:

```
{#templates/cms/default.html#}
{% extends "base.html" %}
{% load cms_tags %}

{% block title %}{% page_attribute "page_title" %} - {% endblock
%}

{% block meta_tags %}
    <meta name="description" content="{% page_attribute meta_
description %}" />
{% endblock %}

{% block content %}
    <h1>{% page_attribute "page_title" %}</h1>
    <div class="row">
        <div class="col-md-8">
            {% placeholder main_content %}
        </div>
        <div class="col-md-4">
            {% placeholder sidebar %}
        </div>
    </div>
{% endblock %}

{#templates/cms/start.html#}
{% extends "base.html" %}
{% load cms_tags %}
```

```
{% block meta_tags %}
    <meta name="description" content="{% page_attribute meta_
description %}" />
{% endblock %}

{% block content %}
    <!--
    Here we add customized website-specific content like
    slideshows, latest tweets, latest news, latest profiles, etc.
    -->
{% endblock %}
```

3. Lastly, we will set the paths of those two templates in the settings as follows:

```
#settings.py
CMS_TEMPLATES = (
    ("cms/default.html", gettext("Default")),
    ("cms/start.html", gettext("Homepage")),
)
```

How it works...

As usual, the `base.html` template is the main template that is extended by all the other templates. In this template, Django CMS uses the `{% render_block %}` template tag from the `django-sekizai` module to inject CSS and JavaScript into templates to create a toolbar and other administration widgets in the frontend. We are inserting the `{% cms_toolbar %}` template tag at the beginning of the `<body>` section—that's where the toolbar will be placed. We use the `{% show_menu_below_id %}` template tag to render header and footer menus from specific page menu trees. Also, we use the `{% language_chooser %}` template tag to render the language chooser that switches to the same page in different languages.

The `default.html` and `start.html` templates defined in the `CMS_TEMPLATES` setting will be available to choose from when creating a CMS page. In those templates, for each area that needs to have dynamically entered content, add a `{% placeholder %}` template tag when you need page-specific content or `{% static_placeholder %}` when you need content shared among different pages. Logged-in administrators can add content plugins to the placeholders when they switch to the draft mode in the CMS toolbar.

See also

- ▶ The *Arranging the base.html template* recipe in *Chapter 4, Templates and JavaScript*
- ▶ The *Structuring the page menu* recipe

Structuring the page menu

In this recipe, we will discuss some guidelines about defining the structure for the pages of your website.

Getting ready

It is a good practice to set the available languages for your website before creating the structure of your pages (although the Django CMS database structure allows you to add new languages later too). Besides `LANGUAGES`, make sure that you have `CMS_LANGUAGES` set in your settings. The `CMS_LANGUAGES` setting defines which languages should be active for each Django site:

```
#settings.py
# ...
from django.utils.translation import ugettext_lazy as _

LANGUAGES = (
    ("en", _("en")),
    ("de", _("de")),
    ("fr", _("fr")),
    ("lt", _("lt")),
)

CMS_LANGUAGES = {
    "default": {
        "public": True,
        "hide_untranslated": False,
        "redirect_on_fallback": True,
    },
    1: [
        {
            "public": True,
            "code": "en",
            "hide_untranslated": False,
            "name": _("en"),
            "redirect_on_fallback": True,
        },
        {
            "public": True,
            "code": "de",
            "hide_untranslated": False,
```

```

        "name": _("de"),
        "redirect_on_fallback": True,
    },
    {
        "public": True,
        "code": "fr",
        "hide_untranslated": False,
        "name": _("fr"),
        "redirect_on_fallback": True,
    },
    {
        "public": True,
        "code": "lt",
        "hide_untranslated": False,
        "name": _("lt"),
        "redirect_on_fallback": True,
    },
],
}

```

How to do it...

The page navigation is set in tree structures. The first tree is the main tree and, contrary to the other trees, the root node of the main tree is not reflected in the URL structure. The root node of this tree is the home page of the website. Usually, this page has a specific template where you add content aggregated from different apps, for example, a slideshow, actual news, newly registered users, latest tweets, or other latest or featured objects. For a convenient way to render items from different apps, check the *Creating a template tag to include a Queryset in a template* recipe in *Chapter 5, Custom Template Filters and Tags*.

If your website has multiple navigations such as a top navigation, a meta navigation, and a footer navigation, give the root node of each tree an ID in the advanced page settings. This ID will be used in the base template by a template tag, `{% show_menu_below_id %}`. You can read more about this and other menu-related template tags in the official documentation at http://docs.django-cms.org/en/3.0.1/getting_started/resources/navigation.html.

The first tree defines the main structure of the website. If you want to have a page under the root-level URL, for example, `/en/search/` but not `/en/meta/search/`, put this page under the home page. If you don't want a page to be shown in the menu, because it will be linked by an icon or a widget, just hide it from the menu.

The footer navigation usually shows different items than the top navigation with some of the items repeated, for example, the page for developers will be shown only in the footer, whereas the page for the news will be shown in both the header and the footer. For all repeated items, just create a page with redirection to the original page in the main tree. If you want to skip the slug component of the URL of the root of the secondary menu tree, you will need to set the overwrite URL in the advanced settings of the page, for example, the developers page should be under `/en/developers/`, but not `/en/secondary/developers/`.

How it works...

Finally, your page structure will look like this (or more complex):

The screenshot shows the Django administration interface for the 'Pages' section. The title is 'Django administration' and the user is 'admin'. The breadcrumb is 'Home > Cms > Pages'. The main heading is 'Select page to change'. There are buttons for 'Recover deleted pages' and 'Add page'. A search bar and a 'Filter: off' link are also present.

	EN	DE	FR	LT	Menu	Actions	Info
Start page							
News							
Movies							
Music							
Games							
FAQ							
Contact							
Search							
Meta							
Imprint							
Privacy Policy							
Terms of Use							
Footer Navigation							
News							
Developers							
About us							

See also

- ▶ The *Creating a template tag to load a QuerySet in a template* recipe in Chapter 5, *Custom Template Filters and Tags*
- ▶ The *Creating templates for Django CMS* recipe
- ▶ The *Attaching your own navigation* recipe

Converting an app to a CMS app

If you have created an app responsible for some type of objects in your website, for example, movies, you can easily convert it to a Django CMS app and attach it to one of the pages. This will ensure that the root URL of the app is translatable and the menu item is highlighted when clicked. In this recipe, we will convert the `movies` app to a CMS app.

Getting ready

Let's start with the `movies` app that we created in the *Filtering object lists* recipe in Chapter 3, *Forms and Views*.

How to do it...

Follow these steps to convert a usual Django app, `movies`, to a Django CMS app:

1. First of all, remove or comment out the URL configuration of the app, because it will be included by an apphook in Django CMS:

```
#urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, include, url
from django.conf import settings
from django.conf.urls.static import static
from django.contrib.staticfiles.urls import \
    staticfiles_urlpatterns
from django.conf.urls.i18n import i18n_patterns
from django.contrib import admin
admin.autodiscover()

urlpatterns = i18n_patterns("",

    # remove or comment out the inclusion of app's urls

    # url(r"^movies/", include("movies.urls")),
```



```
url(r"^admin/", include(admin.site.urls)),
url(r"^", include("cms.urls")),
)

urlpatterns += staticfiles_urlpatterns()
urlpatterns += static(settings.MEDIA_URL,
                      document_root=settings.MEDIA_ROOT)
```

2. Create a file named `cms_app.py` in the `movies` directory and create `MoviesApphook` there as follows:

```
#movies/cms_app.py
# -*- coding: UTF-8 -*-
from django.utils.translation import ugettext_lazy as _
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

class MoviesApphook(CMSApp):
    name = _("Movies")
    urls = ["movies.urls"]

apphook_pool.register(MoviesApphook)
```

3. Set the newly created apphook in the settings as follows:

```
#settings.py
CMS_APPHOOKS = (
    # ...
    "movies.cms_app.MoviesApphook",
)
```

4. Finally, in all the movie templates, change the first line to extend from the template of the current page instead of `base.html`:

```
{% templates/movies/movies_list.html %}

#Change
{% extends "base.html" %}

#to
{% extends request.current_page.template|default:"cms/default.
html" %}
```

How it works...

Apphooks are the interfaces joining the URL configuration of apps to CMS pages. Apphooks need to extend from `CMSApp` and to define the name, which will be shown in the selection list, and the URL rules. Put the path of the apphook in the `CMS_APPHOOKS` project setting, restart the webserver, and the apphook will appear as one of the applications in the advanced page settings. After selecting an application for a page, you need to restart the server again for URLs to take effect.

The templates of the app should extend the page template if you want them to contain placeholders or attributes of the page, for example, the title or the description meta tag.

See also

- ▶ The *Filtering object lists* recipe in *Chapter 3, Forms and Views*
- ▶ The *Attaching your own navigation* recipe

Attaching your own navigation

Once you have an app hooked into CMS pages, you might also want to add a dynamical branch of navigation into the page tree. In this recipe, we will improve the `movies` app and will add new navigation items under the movies page.

Getting ready

Let's say we have a URL configuration (as shown in the following code) for different lists of movies: editor's picks, commercial movies, and independent movies:

```
#movies/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import url, patterns
from django.shortcuts import redirect

urlpatterns = patterns("movies.views",
    url(r'^$', lambda request: redirect("featured_movie_list")),
    url(r'^editors-picks/$', "movie_list", {"featured": True},
        name='featured_movie_list'),
    url(r'^commercial/$', "movie_list", {"commercial": True},
        name="commercial_movie_list"),
    url(r'^independent/$', "movie_list", {"independent": True},
        name="independent_movie_list"),
    url(r'^(?P<slug>[^/]+)/$', "movie_detail",
        name="movie_detail"),
)
```

How to do it...

Follow these two steps to attach the menu choices Editor's Picks, Commercial Movies, and Independent Movies to the navigational menu under the movies page:

1. Create the `menu.py` file in the `movies` app and add the following `MoviesMenu` class there:

```
movies/menu.py
# -*- coding: UTF-8 -*-
from django.utils.translation import ugettext_lazy as _
from django.core.urlresolvers import reverse
from menus.base import NavigationNode
from menus.menu_pool import menu_pool
from cms.menu_bases import CMSAttachMenu

class MoviesMenu(CMSAttachMenu):
    name = _("Movies Menu")

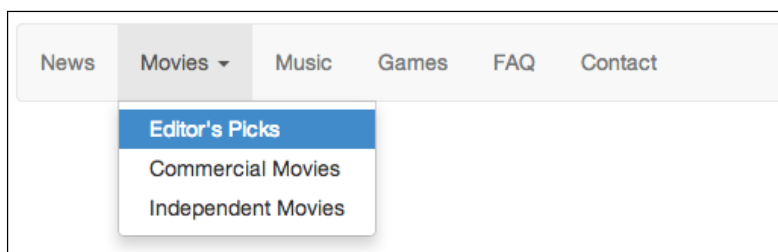
    def get_nodes(self, request):
        nodes = [
            NavigationNode(
                _("Editor's Picks"),
                reverse("featured_movie_list"),
                1,
            ),
            NavigationNode(
                _("Commercial Movies"),
                reverse("commercial_movie_list"),
                2,
            ),
            NavigationNode(
                _("Independent Movies"),
                reverse("independent_movie_list"),
                3,
            ),
        ]
        return nodes

menu_pool.register_menu(MoviesMenu)
```

2. Edit the advanced settings of the movies page and select **Movies Menu**.

How it works...

In the frontend, you will see the new menu items attached to the **Movies** page:



Dynamical menus that are attachable to pages need to extend `CMSAttachMenu`, define the name by which it will be selected, and define the `get_nodes` method, which returns a list of `NavigationNode` objects. The `NavigationNode` class takes three parameters: the title of the menu item, the URL path of the menu item, and the ID of the node. The optional fourth parameter is the ID of the parent node if you want to create a hierarchical dynamical menu. Those IDs can be chosen freely, with the only requirement being they have to be unique in this attached menu.

For other examples of attachable menus, see the official documentation:

http://docs.django-cms.org/en/3.0.1/extending_cms/app_integration.html

See also

- ▶ The *Structuring the page menu* recipe
- ▶ The *Converting an app to a CMS app* recipe

Writing your own CMS plugin

Django CMS comes with a lot of content plugins that you can use in template placeholders, such as the text plugin, flash plugin, picture plugin, and Google map plugin. However, for more structured and better styled content, you would need your own custom plugins, which are not too difficult to create. In this recipe, we will see how to create a new plugin and have a custom layout for its data depending on the chosen template of the page.

Getting ready

Let's create an app named `editorial` and mention it in the `INSTALLED_APPS` setting. Also, we will need the `cms/magazine.html` template created and mentioned in the `CMS_TEMPLATES` setting; you can just duplicate the `cms/default.html` template for that.

How to do it...

To create the `EditorialContent` plugin, follow these steps:

1. In the `models.py` file of the newly created app, add the `EditorialContent` model extending from `CMSPlugin`. The `EditorialContent` model will have these fields: `title`, `subtitle`, `description`, `website`, `image`, `image caption`, and a CSS class:

```
#editorial/models.py
# -*- coding: UTF-8 -*-
import os
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.conf import settings
from django.utils.timezone import now as tz_now
from cms.models import CMSPlugin

def upload_to(instance, filename):
    now = tz_now()
    filename_base, filename_ext = \
        os.path.splitext(filename)
    return 'quotes/%s%s' % (
        now.strftime("%Y/%m/%Y%m%d%H%M%S"),
        filename_ext.lower(),
    )

class EditorialContent(CMSPlugin):
    title = models.CharField(_("Title"), max_length=255)
    subtitle = models.CharField(_("Subtitle"),
                                max_length=255, blank=True)
    description = models.TextField(_("Description"),
                                   blank=True)
    website = models.CharField(_("Website"),
                               max_length=255, blank=True)

    image = models.ImageField(_("Image"), max_length=255,
                              upload_to=upload_to, blank=True)
    image_caption = models.TextField(_("Image Caption"),
                                     blank=True)
```

```

css_class = models.CharField(_("CSS Class"),
                              max_length=255, blank=True)

def __unicode__(self):
    return self.title

class Meta:
    ordering = ["title"]
    verbose_name = _("Editorial content")
    verbose_name_plural = _("Editorial contents")

```

2. In the same app, create a file named `cms_plugins.py` and add a `EditorialContentPlugin` class extending `CMSPluginBase`. This class is a little bit like `ModelAdmin`—it defines the appearance of administration settings for the plugin:

```

#editorial/cms_plugins.py
# -*- coding: utf-8 -*-
import re
from django.utils.translation import ugettext as _
from django.template.loader import select_template
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from models import EditorialContent

class EditorialContentPlugin(CMSPluginBase):
    model = EditorialContent
    name = _("Editorial Content")
    render_template = "cms/plugins/editorial_content.html"

    fieldsets = (
        (_("Main Content"), {
            "fields": ("title", "subtitle",
                      "description", "website"),
            "classes": ["collapse open"]
        }),
        (_("Image"), {
            "fields": ("image", "image_caption"),
            "classes": ["collapse open"]
        }),
        (_("Presentation"), {
            "fields": ("css_class",),
            "classes": ["collapse closed"]
        }),
    )

```

```
def render(self, context, instance, placeholder):
    context.update({
        "object": instance,
        "placeholder": placeholder,
    })
    return context
```

```
plugin_pool.register_plugin(EditorialContentPlugin)
```

3. The `CMS_PLACEHOLDER_CONF` setting defines which plugins go to which placeholders. Also, you can define extra context for the plugins rendered in a specific placeholder. Let's mention `EditorialContentPlugin` for the `main_content` placeholder, and let's also set the `editorial_content_template` context variable for the `main_content` placeholder within the `cms/magazine.html` template:

```
#settings.py
CMS_PLACEHOLDER_CONF = {
    "main_content": {
        "name": gettext("Main Content"),
        "plugins": (
            "EditorialContentPlugin",
            "TextPlugin",
        ),
    },
    "cms/magazine.html main_content": {
        "name": gettext("Magazine Main Content"),
        "plugins": (
            "EditorialContentPlugin",
            "TextPlugin",
        ),
        "extra_context": {
            "editorial_content_template": \
                "cms/plugins/editorial_content/magazine.html",
        }
    },
}
```

4. Then, we will create two templates. One of them will be the `editorial_plugin.html` template. It checks whether the `editorial_content_template` context variable exists, and if it does, it includes it. Otherwise, it shows the default layout for editorial content:

```
{#templates/cms/plugins/editorial_plugin.html#}
{% load i18n %}
```

```

{% if editorial_content_template %}
    {% include editorial_content_template %}
{% else %}
    <div class="item{% if object.css_class %} {{ object.css_class }}{% endif %}">
        <!-- editorial content for non-specific placeholders -->
        <div class="img">
            {% if object.image %}
                
            {% endif %}
            {% if object.image_caption %}<p class="caption">{{ object.image_caption|removetags:"p" }}</p>
            {% endif %}
        </div>
        <h3><a href="{{ object.website }}">{{ object.title }}</a></h3>
        <h4>{{ object.subtitle }}</h4>
        <div class="description">{{ object.description|safe }}</div>
    </div>
{% endif %}

```

5. The second template is a specific template for the EditorialContent plugin within the cms/magazine.html template. There's nothing too fancy here, just an additional CSS class for the container to make the plugin stand out:

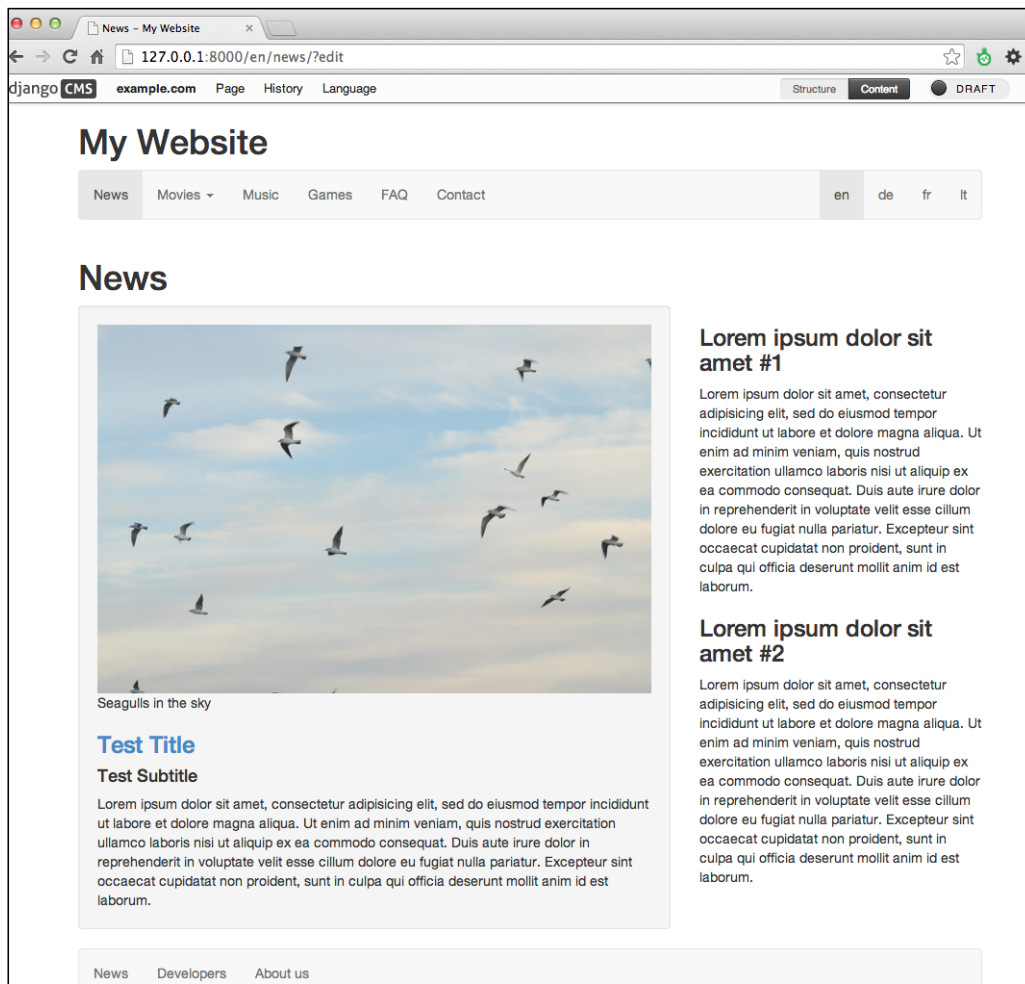
```

{#templates/cms/plugins/editorial_plugin/magazine.html#}
{% load i18n %}
<div class="well item{% if object.css_class %} {{ object.css_class }}{% endif %}">
    <!-- editorial content for non-specific placeholders -->
    <div class="img">
        {% if object.image %}
            
        {% endif %}
        {% if object.image_caption %}<p class="caption">{{ object.image_caption|removetags:"p" }}</p>
        {% endif %}
    </div>
    <h3><a href="{{ object.website }}">{{ object.title }}</a></h3>
    <h4>{{ object.subtitle }}</h4>
    <div class="description">{{ object.description|safe }}</div>
</div>

```


How it works...

If we choose the `cms/magazine.html` template for the news page, we can add the `EditorialContent` plugin there and it will be rendered using the specific template we created in the final step:



See also

- ▶ The *Creating templates for Django CMS* recipe
- ▶ The *Structuring the page menu* recipe

Adding new fields to the CMS page

CMS pages have several multilingual fields such as title, slug, menu title, and page title, as well as the description meta tag and overwrite URL. Also, they have several common non-language-specific fields such as template, ID used in template tags, attached application, and attached menu. But that might not be enough for more complex websites. Gladly, Django CMS features a manageable mechanism to add new database fields for CMS pages. In this recipe, we will show you how to add fields for CSS classes for the navigational menu items and for the page body.

Getting ready

Let's create the `cms_extensions` app and put it under `INSTALLED_APPS` in the settings.

How to do it...

To create a CMS page extension with the CSS class fields for the navigational menu items and for the page body, follow these steps:

1. In the `models.py` file, create a class named `CSSExtension` extending `PageExtension`, and put fields for the menu item's CSS class and for the `<body>` CSS class there:

```
#cms_extensions/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from cms.extensions import PageExtension
from cms.extensions.extension_pool import extension_pool

MENU_ITEM_CSS_CLASS_CHOICES = (
    ('featured', '.featured'),
)
BODY_CSS_CLASS_CHOICES = (
    ('serious', '.serious'),
    ('playful', '.playful'),
)

class CSSExtension(PageExtension):
    menu_item_css_class = models.CharField(
        _("Menu Item CSS Class"),
        max_length=200,
        blank=True,
```

```
        choices=MENU_ITEM_CSS_CLASS_CHOICES,
    )
    body_css_class = models.CharField(
        _("Body CSS Class"),
        max_length=200,
        blank=True,
        choices=BODY_CSS_CLASS_CHOICES,
    )
```

```
extension_pool.register(CSSExtension)
```

2. In the `admin.py` file, let's add administration options for the `CSSExtension` model that we just created:

```
#cms_extensions/admin.py
# -*- coding: UTF-8 -*-
from django.contrib import admin
from cms.extensions import PageExtensionAdmin
from models import CSSExtension

class CSSExtensionAdmin(PageExtensionAdmin):
    pass

admin.site.register(CSSExtension, CSSExtensionAdmin)
```

3. Then, we need to show the CSS extension in the toolbar for each page. This can be done by putting the following code in the `cms_toolbar.py` file in the app:

```
#cms_extensions/cms_toolbar.py
# -*- coding: UTF-8 -*-
from cms.api import get_page_draft
from cms.toolbar_pool import toolbar_pool
from cms.toolbar_base import CMSToolbar
from cms.utils import get cms_setting
from cms.utils.permissions import has_page_change_permission
from django.core.urlresolvers import reverse, NoReverseMatch
from django.utils.translation import ugettext_lazy as _
from models import CSSExtension

@toolbar_pool.register
class CSSExtensionToolbar(CMSToolbar):
    def populate(self):
        # always use draft if we have a page
        self.page = \
            get_page_draft(self.request.current_page)
```

```

if not self.page:
    # Nothing to do
    return

# check global permissions
# if CMS_PERMISSIONS is active
if get_cms_setting('PERMISSION'):
    has_global_current_page_change_permission = \
        has_page_change_permission(self.request)
else:
    has_global_current_page_change_permission = \
        False
    # check if user has page edit permission
can_change = self.request.current_page and \
    self.request.current_page.\
    has_change_permission(self.request)
if has_global_current_page_change_permission or \
    can_change:
    try:
        extension = CSSExtension.objects.get(
            extended_object_id=self.page.id
        )
    except CSSExtension.DoesNotExist:
        extension = None
    try:
        if extension:
            url = reverse(
                "admin:cms_extensions_\
                "cssextension_change",
                args=(extension.pk,)
            )
        else:
            url = reverse(
                "admin:cms_extensions_\
                "cssextension_add"
            ) + '?extended_object=%s' % \
                self.page.pk
    except NoReverseMatch:
        # not in urls
        pass
    else:
        not_edit_mode = not self.toolbar.edit_mode
        current_page_menu = \
            self.toolbar.get_or_create_menu('page')

```

```
current_page_menu.add_modal_item(
    _('CSS'),
    url=url,
    disabled=not_edit_mode
)
```

This code checks whether the user has permissions to change the current page, and if so, it loads the page menu from the current toolbar and adds a new menu item, CSS, with the link to create or edit `CSSExtension`.

4. As we want to access the CSS extension in the navigation menu to attach a CSS class, we need to create a menu modifier in the `menu.py` file in the same app:

```
#cms_extensions/menu.py
# -*- coding: UTF-8 -*-
from cms.models import Page
from menus.base import Modifier
from menus.menu_pool import menu_pool

class CSSModifier(Modifier):
    def modify(self, request, nodes, namespace, root_id,
               post_cut, breadcrumb):
        if post_cut:
            return nodes
        for node in nodes:
            try:
                page = Page.objects.get(pk=node.id)
            except:
                continue
            try:
                page.cssextension
            except:
                pass
            else:
                node.cssextension = page.cssextension
        return nodes
menu_pool.register_modifier(CSSModifier)
```

5. Then, we add the body CSS class to the `<body>` element in the `base.html` template as follows:

```
{# templates/base.html %}
<body class="{% block bodyclass %}{% endblock %}{% if request.
current_page.cssextension %}{{ request.current_page.cssextension.
body_css_class }}{% endif %}">
```

6. Lastly, we modify the `menu.html` file, which is the default template for the navigation menu, and add the menu item's CSS class as follows:

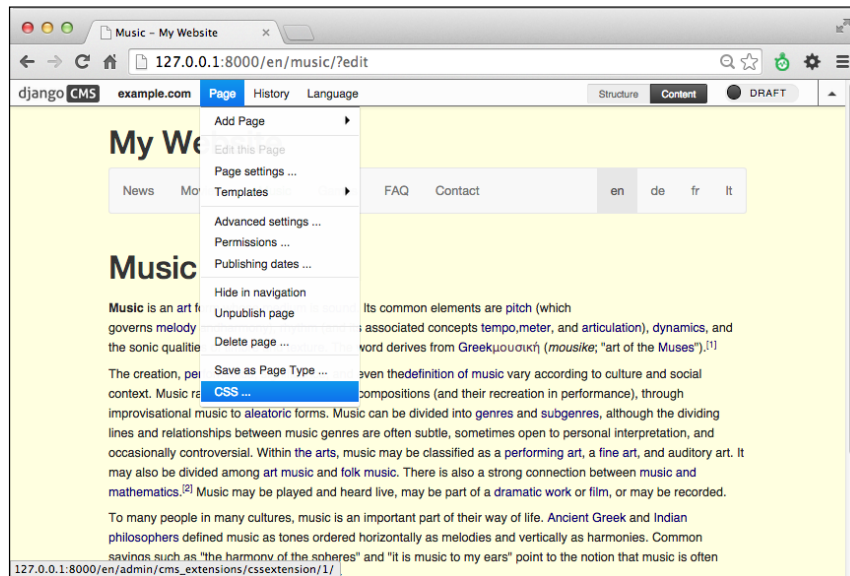
```
{#templates/menu/menu.html#}
{% load i18n menu_tags cache %}

{% for child in children %}
    <li class="{% if child.ancestor %}ancestor{% endif %}
    {% if child.selected %} active{% endif %}{% if child.children
    %} dropdown{% endif %}{% if child.cssextension %} {{ child.
    cssextension.menu_item_css_class }}{% endif %}">
        {% if child.children %}<a class="dropdown-toggle" data-
        toggle="dropdown" href="#">{{ child.get_menu_title }} <span
        class="caret"></span></a>
            <ul class="dropdown-menu">
                {% show_menu from_level to_level extra_inactive
                extra_active template "" "" child %}
            </ul>
        {% else %}
            <a href="{{ child.get_absolute_url }}"><span>{{ child.
            get_menu_title }}</span></a>
        {% endif %}
    </li>
{% endfor %}
```

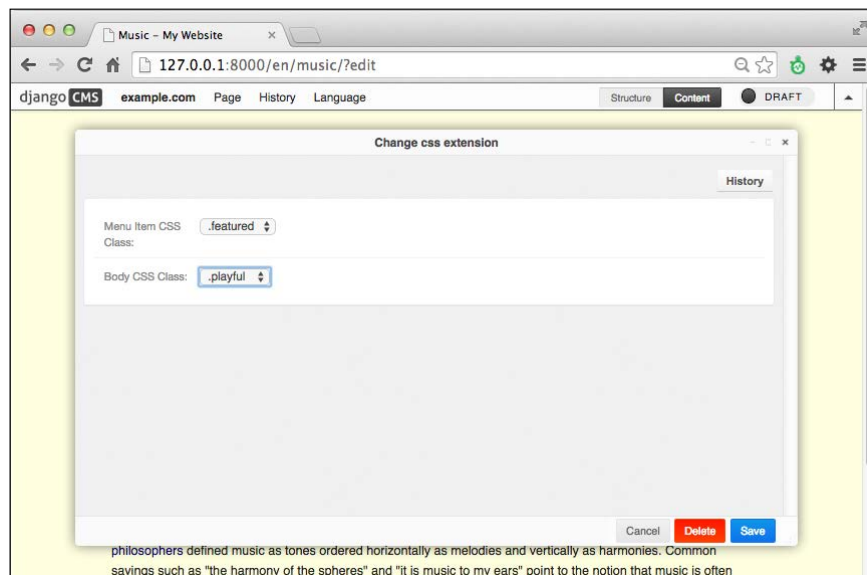
How it works...

The `PageExtension` class is a model mixin with a one-to-one relationship with the `Page` model. To be able to administrate the custom extension model within Django CMS, there is a specific `PageExtensionAdmin` class to extend. Then, in the `cms_toolbar.py` file, we create the `CSSExtensionToolbar` class inheriting from the `CMSToolbar` class to create an item in the Django CMS toolbar. In the `populate` method, we do the general routine to check page permissions and then we add a **CSS** menu item to the toolbar.

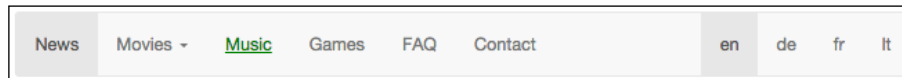
If the administrator has changed permissions for the page, then he/she will see a **CSS** option in the toolbar under the **Page** menu item:



When the administrator clicks on the new **CSS** link, a pop-up window opens and he/she can select CSS classes for the navigation menu item and for the body.



To show a specific CSS class from the **Page** extension in the navigation menu, we need to attach the `CSSExtension` objects to navigation items accordingly. Then, those objects can be accessed in the `menu.html` template as `{{ child.cssextension }}`. In the end, you will have navigation menu items highlighted like the **Music** item shown here (depending on your CSS):



To access the `CSSExtension` object of the current page in a template for the body CSS class, you can use `{{ request.current_page.cssextension }}`.

See also

- ▶ The *Creating templates for Django CMS* recipe

8

Hierarchical Structures

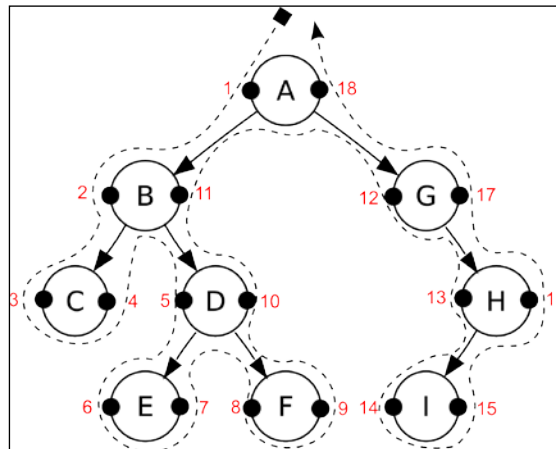
In this chapter, we will cover the following recipes:

- ▶ Creating hierarchical categories
- ▶ Creating a category administration interface with `django-mptt-admin`
- ▶ Creating a category administration interface with `django-mptt-tree-editor`
- ▶ Rendering categories in a template
- ▶ Using a single selection field to choose a category in forms
- ▶ Using a checkbox list to choose multiple categories in forms

Introduction

Whether you build your own forum, threaded comments, or categorization system, there will be a moment when you need to save hierarchical structures in the database. Although the tables of relational databases (such as MySQL or PostgreSQL) are of a flat manner, there is a fast and effective way to store hierarchical structures there. It is called **Modified Preorder Tree Traversal (MPTT)**. MPTT lets you read the tree structures without recursive calls to the database.

I'll explain how MPTT works. Imagine that you lay out your tree horizontally with the root node at the top. Each node in the tree has left and right values. Imagine them as small left and right handles on the left and on the right of the node. Then, you walk (traverse) around the tree counter-clockwise starting from the root node and mark each left or right value you find with a number: 1, 2, 3, and so on. It will look like this:



In the database table of this hierarchical structure, you would have a title, the left value, and the right value for each node.

Now, if you want to get the subtree of node **B** with **2** as the left value and **11** as the right value, you would have to select all the nodes that have a left value between **2** and **11**. They are **C**, **D**, **E**, and **F**.

To get all ancestors of node **D** with **5** as the left value and **10** as the right value, you have to select all the nodes that have a left value less than **5** and a right value of more than **10**. These would be **B** and **A**.

To get the number of descendants for a node, you can use this formula:

$$\text{descendants} = (\text{right} - \text{left} - 1) / 2.$$

So, the number of descendants for node **B** can be calculated like this: $(11 - 2 - 1) / 2 = 4$.

If we wanted to attach node **E** to node **C**, we would have to update the left and right values only for the nodes of their first common ancestor, node **B**.

Similarly, there are other tree-related operations with nodes in MPTT. It might be too complicated to manage this yourself for every hierarchical structure in your project, but luckily there is a Django app called `django-mptt` that handles these algorithms and provides an easy API to handle the tree structures. Django CMS also uses `django-mptt` for web pages. In this chapter, you will learn how to use this helper app.

Creating hierarchical categories

To illustrate how to deal with MPTT, we will create a `movies` app that will have a hierarchical `Category` model and a `Movie` model with a many-to-many relationship with the categories.

Getting ready

To get started, complete the following steps:

1. Install `django-mptt` in your virtual environment using the following command:

```
(myproject_env)$ pip install django-mptt
```
2. Then, create a `movies` app. Add the `movies` app as well as `mptt` to `INSTALLED_APPS` in the settings as follows:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "mptt",
    "movies",
)
```

How to do it...

We will create a hierarchical `Category` model and then create a `Movie` model that will have a many-to-many relationship with the categories, as follows:

1. Open the `models.py` file and add a `Category` model that extends `mptt.models.MPTTModel` and `CreationModificationDateMixin` that we defined in *Chapter 2, Database Structures*. In addition to the fields coming from the mixins, the `Category` model will have to have a parent field of the `TreeForeignKey` type and a `title` field:

```
#movies/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from utils.models import CreationModificationDateMixin
from mptt.models import MPTTModel
from mptt.fields import TreeForeignKey, TreeManyToManyField

class Category(MPTTModel, CreationModificationDateMixin):
    parent = TreeForeignKey("self", blank=True, null=True)
    title = models.CharField(_("Title"), max_length=200)
```

```
def __unicode__(self):
    return self.title

class Meta:
    ordering = ["tree_id", "lft"]
    verbose_name = _("Category")
    verbose_name_plural = _("Categories")
```

2. Then, create the `Movie` model that extends `CreationModificationDateMixin` and also include a `title` and `categories` field of the `TreeManyToManyField` type:

```
class Movie(CreationModificationDateMixin):
    title = models.CharField(_("Title"), max_length=255)
    categories = TreeManyToManyField(Category,
                                     verbose_name=_("Categories"))

    def __unicode__(self):
        return self.title

    class Meta:
        verbose_name = _("Movie")
        verbose_name_plural = _("Movies")
```

How it works...

The `MPTTModel` mixin will add the `tree_id`, `lft`, `right`, and `level` fields to the `Category` model. The `tree_id` field is used because you can have multiple trees in the database table. In fact, each root category is saved in a separate tree. The `lft` and `right` fields store the left and right values used in MPTT algorithms. The `level` field stores the node's depth in the tree. The root node will be level 0.

Besides new fields, the `MPTTModel` mixin adds methods to navigate through the tree structure somewhat like you would navigate through DOM elements using JavaScript. These methods are listed as follows:

- ▶ If you want to get the ancestors of a category, use the following code:

```
ancestor_categories = category.get_ancestors(ascending=False,
                                             include_self=False)
```

The `ascending` parameter defines from which direction to read the nodes (the default is `False`). The `include_self` parameter defines whether to include the category itself in the `QuerySet` (the default is `False`).

- ▶ To get just the root category, use the following code:

```
root = category.get_root()
```

- ▶ If you want to get the direct children of a category, use the following code:

```
children = category.get_children()
```

- ▶ To get all descendants of a category, use the following code:

```
descendants = category.get_descendants(include_self=False)
```

Here, again, the `include_self` parameter defines whether or not to include the category itself in `QuerySet`.

- ▶ If you want to get the descendant count without querying the database, use the following code:

```
descendants_count = category.get_descendant_count()
```

- ▶ To get all siblings, call the following methods:

```
siblings = category.get_siblings(include_self=False)
```

Root categories are considered to be siblings of other root categories.

- ▶ To get just the previous and next siblings, call the following methods:

```
previous_sibling = category.get_previous_sibling()
```

```
next_sibling = category.get_next_sibling()
```

- ▶ Also, there are methods to check whether the category is a root, child, or leaf:

```
category.is_root_node()
```

```
category.is_child_node()
```

```
category.is_leaf_node()
```

All these methods can be used either in the views, templates, or management commands. If you want to manipulate the tree structure, you can also use the `insert_at()` and `move_to()` methods. In that case, you can read about them and also about the tree manager methods at <http://django-mptt.github.io/django-mptt/models.html>.

In the preceding models, we used `TreeForeignKey` and `TreeManyToManyField`. These are like `ForeignKey` and `ManyToManyField` except that they show the choices indented in hierarchies in the administration interface.

Also, note that in the `Meta` class of the `Category` model, we order the categories by `tree_id` and then by the `lft` value to show the categories naturally in the tree structure.

See also

- ▶ The *Creating a model mixin to handle creation and modification dates* recipe in *Chapter 2, Database Structures*
- ▶ The *Structuring the page menu* recipe in *Chapter 7, Django CMS*
- ▶ The *Creating a category administration interface with django-mptt-admin* recipe

Creating a category administration interface with django-mptt-admin

The `django-mptt` app comes with a simple model administration mixin that allows you to create the tree structure and list it with indentation. In order to reorder trees, you need to either create this functionality yourself or you can use a third-party solution. Currently, there are two apps that can help you to create a draggable administration interface for hierarchical models. One of them is `django-mptt-admin`. Let's have a look at it in this recipe.

Getting ready

First, we need to have the `django-mptt-admin` app installed by performing the following steps:

1. To start, install the app in your virtual environment using the following command:

```
(myproject_env)$ pip install django-mptt-admin
```
2. Then, put it in `INSTALLED_APPS` in the settings, as follows:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "django_mptt_admin"
)
```

How to do it...

Create an administration interface for the `Category` model that extends `DjangoMpttAdmin` instead of `admin.ModelAdmin`, as follows:

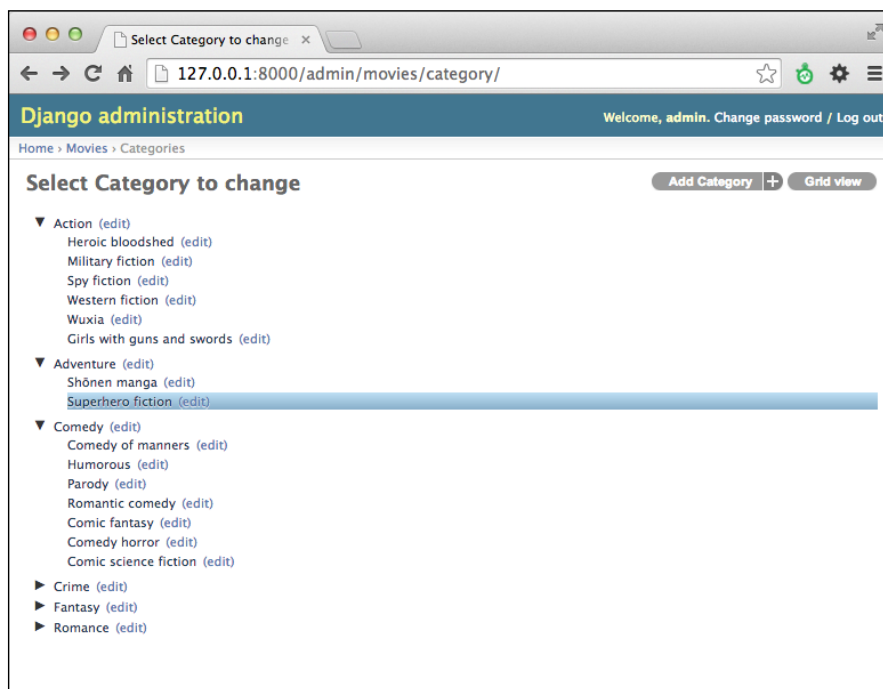
```
#movies/admin.py
# -*- coding: UTF-8 -*-
from django.contrib import admin
from django_mptt_admin.admin import DjangoMpttAdmin
from models import Category
```

```
class CategoryAdmin(DjangoMpttAdmin):
    list_display = ["title", "created", "modified"]
    list_filter = ["created"]

admin.site.register(Category, CategoryAdmin)
```

How it works...

The administration interface for the categories will have two modes: **tree view** and **grid view**. The tree view looks like this:



The tree view uses the `jqTree` jQuery library for node manipulation. You can expand and collapse categories for a better overview. To reorder them or to change the dependencies, you can drag-and-drop the titles in this list view. Note that any usual list-related settings such as `list_display` or `list_filter` will be ignored.

If you want to filter categories, sort or filter them by a specific field, or apply admin actions, you can switch to the grid view, which shows the default category change list.

See also

- ▶ The *Creating hierarchical categories* recipe
- ▶ The *Creating a category administration interface with django-mptt-tree-editor* recipe

Creating a category administration interface with django-mptt-tree-editor

If you want to use the common functionality of the change list in your administration interface, such as columns, admin actions, editable fields, or filters, as well as manipulate the tree structure in the same view, you need to use another third-party app called `django-mptt-tree-editor`. Let's see how to do that.

Getting ready

First, we need to have the `django-mptt-tree-editor` app installed. Perform the following steps:

1. To start, install the app in your virtual environment using the following command:

```
(myproject_env)$ pip install django-mptt-tree-editor
```
2. Then, put it in `INSTALLED_APPS` in the settings, as follows:

```
#settings.py
INSTALLED_APPS = (
    # ...
    "mptt_tree_editor"
)
```

How to do it...

Create an administration interface for the `Category` model that extends `TreeEditor` instead of `admin.ModelAdmin`. Make sure you add `indented_short_title` and `actions_column` at the beginning of the `list_display` setting, as follows:

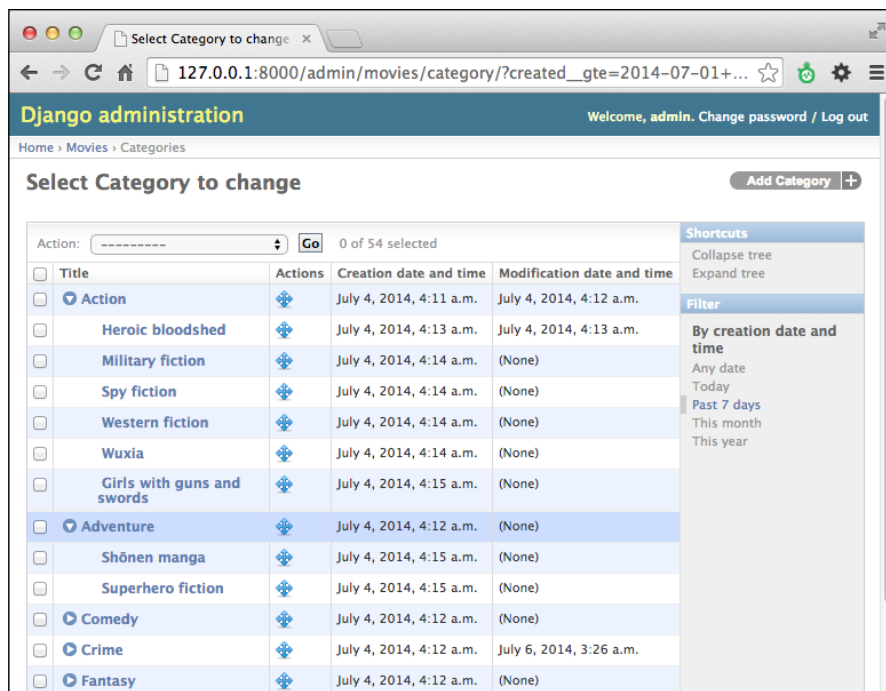
```
#movies/admin.py
# -*- coding: UTF-8 -*-
from django.contrib import admin
from mptt_tree_editor.admin import TreeEditor
from models import Category
```

```
class CategoryAdmin(TreeEditor):
    list_display = ["indented_short_title", "actions_column",
                   "created", "modified"]
    list_filter = ["created"]

admin.site.register(Category, CategoryAdmin)
```

How it works...

The administration interface for your categories now looks like this:



The category administration interface allows you to expand or collapse categories. The `indented_short_title` column will either return the indented short title from the `short_title` method of the category (if there is one) or the indented unicode representation of the category. The column defined as `actions_column` will be rendered as a handle to reorder or restructure categories by dragging-and-dropping them. As the handle is detached from the category title, it might feel weird to work with it. But as you can see, you have the possibility to use all list-related features of the default Django administration interface in the same view.

In `django-mptt-tree-editor`, the tree-editing functionality was ported from FeinCMS, another content management system made with Django.

See also

- ▶ The *Creating hierarchical categories* recipe
- ▶ The *Creating a category administration interface with django-mptt-admin* recipe

Rendering categories in a template

Once you have created categories in your app, you need to display them hierarchically in a template. The easiest way to do that is to use the `{% recursetree %}` template tag from the `django-mptt` app. We will show you how to do that in this recipe.

Getting ready

Make sure that you have the `Category` model created and some categories entered in the database.

How to do it...

Pass `QuerySet` of your hierarchical categories to the template and then use the `{% recursetree %}` template tag as follows:

1. Create a view that loads all categories and passes them to a template:

```
#movies/views.py
# -*- coding: UTF-8 -*-
from django.shortcuts import render
from models import Category

def movie_category_list(request):
    context = {
        "categories": Category.objects.all(),
    }
    return render(request,
        "movies/movie_category_list.html", context)
```

2. Create a template with the following content:

```
{% templates/movies/movie_category_list.html %}
{% extends "base.html" %}
{% load mptt_tags %}

{% block content %}
<ul class="root">
```

```

{% recursetree categories %}
  <li>
    {{ node.title }}
    {% if not node.is_leaf_node %}
      <ul class="children">
        {{ children }}
      </ul>
    {% endif %}
  </li>
{% endrecursetree %}
</ul>
{% endblock %}

```

3. Create a URL rule to show the view.

How it works...

The template will be rendered as nested lists:

- Action
 - ◊ Heroic bloodshed
 - ◊ Military fiction
 - ◊ Spy fiction
 - ◊ Western fiction
 - ◊ Wuxia
 - ◊ Girls with guns and swords
- Adventure
 - ◊ Shōnen manga
 - ◊ Superhero fiction
- Comedy
 - ◊ Comedy of manners
 - ◊ Humorous
 - ◊ Parody
 - ◊ Romantic comedy
 - ◊ Comic fantasy
 - ◊ Comedy horror
 - ◊ Comic science fiction

The `{% recursetree %}` block template tag takes `QuerySet` of the categories and renders the list using the template content within the tag. There are two special variables used here: `node` and `children`. The `node` variable is an instance of the `Category` model. You can use its fields or methods such as `{{ node.get_descendant_count }}`, `{{ node.level }}`, or `{{ node.is_root }}` to add specific CSS classes or HTML5 data attributes for JavaScript. The second variable, `children`, defines where to place the children of the current category.

There's more...

If your hierarchical structure is very complex with more than 20 depth levels, it is recommended to use the non-recursive template filter, `tree_info`. For more information on how to do this, please refer to the official documentation at <http://django-mptt.github.io/django-mptt/templates.html#tree-info-filter>.

See also

- ▶ The *Using HTML5 data attributes* recipe in *Chapter 4, Templates and JavaScript*
- ▶ The *Creating hierarchical categories* recipe
- ▶ The *Using a single selection field to choose a category in forms* recipe

Using a single selection field to choose a category in forms

What happens if you want to show category selection in a form? How will the hierarchy be presented? In `django-mptt`, there is a special form field, `TreeNodeChoiceField`, that you can use to show the hierarchical structures in a selected field. Let's have a look at how to do that.

Getting ready

We will start with the `movies` app that we defined in the previous recipes.

How to do it...

Let's create a form with the `category` field and then show it in a view:

1. In the `forms.py` file of the app, create a form with a `category` field as follows:

```
#movies/forms.py
# -*- coding: UTF-8 -*-
from django import forms
```

2. Then, create a URL rule, view, and template to show that form.

How it works...

The category selection will look like this:

Category

Action

Heroic bloodshed

Military fiction

Spy fiction

Western fiction

Wuxia

Girls with guns and swords

Adventure

Shōnen manga

Superhero fiction

Comedy

Comedy of manners

Humorous

Parody

Romantic comedy

Comic fantasy

Comedy horror

Comic science fiction

Crime

`TreeNodeChoiceField` acts like `ModelChoiceField`, but shows hierarchical choices as indented. By default, `TreeNodeChoiceField` represents each deeper level prefixed by three dashes, ---. In our example, we change the level indicator to be four non-breakable spaces (HTML entities ` `;) by passing the `level_indicator` parameter to the field. To ensure that the non-breakable spaces aren't escaped, we use the `mark_safe` function.

See also

- [The Using a checkbox list to choose multiple categories in forms recipe](#)

Using a checkbox list to choose multiple categories in forms

When more than one category needs to be selected in a form, you can use the multiple selection field, `TreeNodeMultipleChoiceField`, provided by `django-mptt`. However, multiple selection fields are very user-unfriendly from the GUI point of view, because the user needs to scroll and hold the control keys while clicking to make multiple choices. That's really awful. A much better way is to provide a checkbox list to choose the categories. In this recipe, we will create a field that allows you to show indented checkboxes in the form.

Getting ready

We will start with the `movies` app that we defined in the previous recipes, and also the `utils` app that you should have in your project.

How to do it...

To render an indented list of categories with checkboxes, create and use a new form field named `MultipleChoiceTreeField`, and also create an HTML template for this field. The specific template will be passed to the `crispy forms` layout in the form. Perform the following steps:

1. In the `utils` app, add a `fields.py` file and create a form field named `MultipleChoiceTreeField` that extends `ModelMultipleChoiceField`, as follows:

```
#utils/fields.py
# -*- coding: utf-8 -*-
from django import forms

class MultipleChoiceTreeField(forms.ModelMultipleChoiceField):
    widget = forms.CheckboxSelectMultiple
    def label_from_instance(self, obj):
        return obj
```

2. Use the new field with the categories to choose from in the form for movie creation. Also, in the form layout, pass a custom template to the `categories` field as follows:

```
#movies/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from django.utils.translation import ugettext_lazy as _
from crispy_forms.helper import FormHelper
from crispy_forms import layout, bootstrap
from utils.fields import MultipleChoiceTreeField
from models import Movie, Category

class MovieForm(forms.ModelForm):
    categories = MultipleChoiceTreeField(
        label=_("Categories"),
        required=False,
        queryset=Category.objects.all(),
    )
    class Meta:
        model = Movie

    def __init__(self, *args, **kwargs):
        super(MovieForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.form_action = ""
        self.helper.form_method = "POST"
        self.helper.layout = layout.Layout(
            layout.Field("title"),
            layout.Field(
                "categories",
                template="utils/" +
                "checkbox_select_multiple_tree.html"
            ),
            bootstrap.FormActions(
                layout.Submit("submit", _("Save")),
            )
        )
```

3. Create a template for a Bootstrap-style checkbox list:

```
{#templates/utils/checkbox_select_multiple_tree.html#}
{% load crispy_forms_filters %}
{% load l10n %}

<div id="div_{{ field.auto_id }}" class="form-group{% if wrapper_
class %} {{ wrapper_class }}{% endif %}{% if form_show_errors%}
{% if field.errors %} has-error{% endif %}{% endif %}{% if field.
css_classes %} {{ field.css_classes }}{% endif %}">
```



```

        {% if field.label and form_show_labels %}
            <label for="{{ field.id_for_label }}" class="control-label
            {{ label_class }}" {% if field.field.required %} requiredField{%
            endif %}">
                {{ field.label|safe }}{% if field.field.required
                %}<span class="asteriskField">*</span>{% endif %}
            </label>
        {% endif %}
        <div class="controls {{ field_class }}" {% if flat_attrs %} {{
        flat_attrs|safe }}{% endif %}>
            {% include 'bootstrap3/layout/field_errors_block.html' %}

            {% for choice_value, choice_instance in field.field.
            choices %}
                <label class="checkbox{% if inline_class %}-{{ inline_
                class }}" {% endif %} level-{{ choice_instance.level }}">
                    <input type="checkbox" {% if choice_value in
                    field.value or choice_value|stringformat:"s" in field.value or
                    choice_value|stringformat:"s" == field.value|stringformat:"s" %}
                    checked="checked" {% endif %} name="{{ field.html_name }}" id="id_
                    {{ field.html_name }}_{{ forloop.counter }}" value="{{ choice_
                    value|unlocalize }}" {{ field.field.widget.attrs|flatatt }}>{{
                    choice_instance }}
                </label>
            {% endfor %}
            {% include "bootstrap3/layout/help_text.html" %}
        </div>
    </div>

```

4. Create a URL rule, view, and template to show the form with the `{% crispy %}` template tag. To see how to use this template tag, please refer to the *Creating a form layout with django-crispy-forms* recipe in *Chapter 3, Forms and Views*.
5. Lastly, add a rule to your CSS file to indent labels with classes, `.level-0`, `.level-1`, `.level-2`, and so on, by setting the `margin-left` parameter, as follows:

```

#style.css
.level-0 {
    margin-left: 0;
}
.level-1 {
    margin-left: 20px;
}
.level-2 {
    margin-left: 40px;
}

```

How it works...

As a result, we get this form:

The screenshot shows a web browser window titled 'My Website' with the address bar displaying '127.0.0.1:8000/movies/add/'. The main heading is 'Add Movie'. Below it is a text input field for 'Title*' containing 'Casino Royale'. Under the 'Categories' section, there are several checkboxes. The checked categories are: Action, Spy fiction, Adventure, and Superhero fiction. The unchecked categories are: Heroic bloodshed, Military fiction, Western fiction, Wuxia, Girls with guns and swords, Shōnen manga, Comedy, Comedy of manners, Humorous, and Parody.

Contrary to the default behavior of Django, which hardcodes field generation in Python code, the `django-crispy-forms` app uses templates to render fields. You can browse them under `crispy_forms/templates/bootstrap3` and, when necessary, you can copy some of them to an analogous path in your project's template directory and overwrite them.

In our movie creation form, we are passing a custom template for the categories field that will add the `.level-*` CSS classes to the `<label>` tag wrapping the checkboxes. One problem with the normal `CheckboxSelectMultiple` widget is that when rendered, it only uses choice values and choice texts, and in our case, we need other properties of the category like the depth level. To solve this, we created a custom form field named `MultipleChoiceTreeField`, which extends `ModelMultipleChoiceField` and overrides the `label_from_instance` method to return the category itself instead of its unicode representation. The template for the field looks complicated, but it is just the combination of a common field template (`crispy_forms/templates/bootstrap3/field.html`) and a multiple checkbox field template (`crispy_forms/templates/bootstrap3/layout/checkboxselectmultiple.html`) with all the necessary Bootstrap 3 markup. We just made a slight modification to add the `.level-*` CSS classes.

See also

- ▶ The *Creating a form layout with django-crispy-forms* recipe in *Chapter 3, Forms and Views*
- ▶ The *Using a single selection field to choose a category in forms* recipe

9

Data Import and Export

In this chapter, we will cover the following recipes:

- ▶ Importing data from a local CSV file
- ▶ Importing data from a local Excel file
- ▶ Importing data from an external JSON file
- ▶ Importing data from an external XML file
- ▶ Creating filterable RSS feeds
- ▶ Using Tastypie to provide data to third parties

Introduction

There are times when your data needs to be transported from a local format to the database, imported from external resources, or provided to third parties. In this chapter, we will have a look at some practical examples of how to write management commands and APIs to do that.

Importing data from a local CSV file

Comma-separated values (CSV) are probably the simplest way to store tabular data in a text file. In this recipe, we will create a management command that imports data from CSV to a Django database. We will need a CSV list of movies with a title, URL, and release year. You can easily create such a file with Excel, Calc, or another spreadsheet application.

Getting ready

Create a `movies` app with the `Movie` model containing these fields: `title`, `url`, and `release_year`. Place it under `INSTALLED_APPS` in the settings.

How to do it...

Follow these steps to create and use a management command that imports movies from a local CSV file:

1. In the `movies` app, create a `management` directory and then a `commands` directory inside the new `management` directory. Put the empty `__init__.py` files in both new directories to make them Python packages.
2. Add an `import_movies_from_csv.py` file there with the following content:

```
#movies/management/commands/import_movies_from_csv.py
# -*- coding: UTF-8 -*-
import csv
from django.core.management.base import BaseCommand
from django.core.management.base import CommandError
from django.utils.encoding import smart_str
from movies.models import Movie
SILENT, NORMAL, VERBOSE, VERY_VERBOSE = 0, 1, 2, 3

class Command(BaseCommand):
    args = "<file_path>"
    help = "Imports movies from a local CSV file. "\
          "Expects title, URL, and release year."

    def handle(self, *args, **options):
        verbosity = int(options.get("verbosity", NORMAL))
        if args:
            file_path = args[0]
        else:
            raise CommandError("Pass a path to a CSV file")

        if verbosity >= NORMAL:
            print "=== Movies imported ==="

        with open(file_path) as f:
            reader = csv.reader(f)
            for title, url, release_year in reader:
                movie, created = Movie.objects.\
                    get_or_create(
                        title=title,
                        url=url,
                        release_year=release_year,
                    )
                if verbosity >= NORMAL:
                    print " - %s" % smart_str(movie.title)
```

3. To run the import, call this method in the command line:

```
(myproject_env)$ python manage.py import_movies_from_csv \
data/movies.csv
```

How it works...

For a management command, we need to create a `Command` class deriving from `BaseCommand` and overwriting the `handle` method. The `args` parameter defines what arguments should be passed. The `help` parameter defines the help text for the management command.

At the beginning of the `handle` method, the `verbosity` argument is checked. Verbosity defines how verbose the command is, from 0 not giving any output to the command-line tool to 2 being very verbose. You can pass this parameter to the command like this:

```
(myproject_env)$ python manage.py import_movies_from_csv movies.csv \
--verbosity=0
```

Then, we also expect the filename as the first positional argument. We open the given file and pass its pointer to `csv.reader`. Then, for each line in the file, we create a new `Movie` object.



If you want to debug the errors of a management command while developing it, pass the `--traceback` parameter to it. If an error happens, you will see the full stack trace of the problem.

See also

- *The Importing data from a local Excel file recipe*

Importing data from a local Excel file

Another popular format to store tabular data is an Excel sheet. In this recipe, we will import movies from a file of this format.

Getting ready

Let's start with the `movies` app that you created in the previous recipe. Install the `xlrd` package for reading Excel files, as follows:

```
(project_env)$ pip install xlrd
```

How to do it...

Follow these steps to create and use a management command that imports movies from a local XLS file:

1. Once again, you need the management package with the `commands` package inside your `movies` app.
2. Add the `import_movies_from_xls.py` file there with the following contents:

```
#movies/management/commands/import_movies_from_xls.py
# -*- coding: UTF-8 -*-
import xlrd
from django.core.management.base import BaseCommand
from django.core.management.base import CommandError
from django.utils.encoding import smart_str
from movies.models import Movie
SILENT, NORMAL, VERBOSE, VERY_VERBOSE = 0, 1, 2, 3

class Command(BaseCommand):
    args = "<file_path>"
    help = "Imports movies from a local XLS file. "\
          "Expects title, URL, and release year."

    def handle(self, *args, **options):
        verbosity = int(options.get("verbosity", NORMAL))
        if args:
            file_path = args[0]
        else:
            raise CommandError("Pass a path to a XLS file")

        wb = xlrd.open_workbook(file_path)
        sh = wb.sheet_by_index(0)

        if verbosity >= NORMAL:
            print "=== Movies imported ==="

        for rownum in xrange(sh.nrows):
            # let's skip the column captions
            if rownum > 0:
                (title, url, release_year) = \
                    sh.row_values(rownum)
                movie, created = Movie.objects.\
                    get_or_create(
                        title=title,
                        url=url,
                        release_year=release_year,
                    )
                if verbosity >= NORMAL:
                    print " - %s" % smart_str(movie.title)
```

3. To run the import, call this in the command line:

```
(myproject_env)$ python manage.py import_movies_from_xls \
data/movies.xls
```

How it works...

The principle of importing from an XLS file is the same as with CSV. We open the file, read it row by row, and create the `Movie` objects out of the provided data.

Excel files are workbooks containing sheets as different tabs.

We are using the `xlrd` library to open a file passed as a positional argument to the command. Then, we read the first sheet from the workbook.

Lastly, we read the rows one by one (except the first row with the column titles) and create the `Movie` objects out of them.

See also

- The *Importing data from a local CSV file* recipe

Importing data from an external JSON file

Music website Last.fm has an API under the `http://ws.audioscrobbler.com/` domain that you can use to read out albums, artists, tracks, events, and more. The API allows you to use either JSON or XML format. In this recipe, we will import the top tracks tagged `disco` using the JSON format.

Getting ready

Follow these steps in order to import data in JSON format from Last.fm:

1. To use Last.fm, you need to register and get an API key. The API key can be created at `http://www.last.fm/api/accounts`.
2. The API key has to be set in the settings as `LAST_FM_API_KEY`.
3. Also, install the `requests` library to your virtual environment using the following command:

```
(myproject_env)$ pip install requests
```


4. Let's check out what the structure of the JSON endpoint is
(http://ws.audioscrobbler.com/2.0/?method=tag.gettoptracks&tag=disco&api_key=xxx&format=json):

```
{
  "toptracks": {
    "track": [
      {
        "name": "Get Lucky (feat. Pharrell Williams)",
        "duration": "102",
        "mbid": "",
        "url": "http://www.last.fm/music/Daft+Punk/_/Get+Lucky
+ (feat.+Pharrell+Williams)",
        "streamable": {
          "#text": "0",
          "fulltrack": "0"
        },
        "artist": {
          "name": "Daft Punk",
          "mbid": "056e4f3e-d505-4dad-8ec1-d04f521cbb56",
          "url": "http://www.last.fm/music/Daft+Punk"
        },
        "image": [
          {
            "#text": "http://userserve-ak.last.fm/
serve/34s/92459761.png",
            "size": "small"
          },
          {
            "#text": "http://userserve-ak.last.
fm/serve/64s/92459761.png",
            "size": "medium"
          },
          {
            "#text": "http://userserve-ak.last.fm/
serve/126/92459761.png",
            "size": "large"
          },
          {
            "#text": "http://userserve-ak.last.fm/
serve/300x300/92459761.png",
            "size": "extralarge"
          }
        ]
      }
    ]
  }
}
```

```

        "@attr": {
            "rank": "2"
        }
    },
    ...
],
"@attr": {
    "tag": "Disco",
    "page": "1",
    "perPage": "50",
    "totalPages": "20",
    "total": "1000"
}
}
}

```

What we want to read out is the track name, artist, URL, and medium-sized image.

How to do it...

Follow these steps to create a `Track` model and then to create a management command that imports top tracks from Last.fm to the database:

1. Let's create a music app with a simple `Track` model there, as follows:

```

#music/models.py
# -*- coding: UTF-8 -*-
import os
from django.utils.translation import ugettext_lazy as _
from django.db import models
from django.utils.text import slugify

def upload_to(instance, filename):
    filename_base, filename_ext = \
        os.path.splitext(filename)
    return "tracks/%s--%s%s" % (
        slugify(instance.artist),
        slugify(instance.name),
        filename_ext.lower(),
    )

class Track(models.Model):
    name = models.CharField(_("Name"), max_length=250)
    artist = models.CharField(_("Artist"), max_length=250)
    url = models.URLField(_("URL"))

```

```
image = models.ImageField(_("Image"),
                           upload_to=upload_to, blank=True, null=True)

class Meta:
    verbose_name = _("Track")
    verbose_name_plural = _("Tracks")

def __unicode__(self):
    return u"%s - %s" % (self.artist, self.name)
```

2. Then, create a management command with the handle method as follows:

```
#music/management/commands/import_music_from_lastfm_as_json.py
# -*- coding: UTF-8 -*-
import os
import requests
from StringIO import StringIO
from django.core.management.base import BaseCommand
from django.core.management.base import CommandError
from django.utils.encoding import smart_str, force_unicode
from django.conf import settings
from django.core.files import File
from music.models import Track

SILENT, NORMAL, VERBOSE, VERY_VERBOSE = 0, 1, 2, 3

class Command(BaseCommand):
    help = "Imports top tracks from Last.fm as JSON."

    def handle(self, *args, **options):
        self.verbosity = int(options.get("verbosity",
                                          NORMAL))

        r = requests.get(
            "http://ws.audioscrobbler.com/2.0/",
            params={
                "method": "tag.gettoptracks",
                "tag": "disco",
                "api_key": settings.LAST_FM_API_KEY,
                "format": "json",
            }
        )
```

```

response_dict = r.json()
total_pages = int(response_dict["toptracks"]\
["@attr"]["totalPages"])

if self.verbosity >= NORMAL:
    print "=== Tracks imported ==="

self.save_page(response_dict)
for page_number in xrange(2, total_pages + 1):
    r = requests.get(
        "http://ws.audioscrobbler.com/2.0/",
        params={
            "method": "tag.gettoptracks",
            "tag": "disco",
            "api_key": settings.LAST_FM_API_KEY,
            "page": page_number,
        }
    )
    response_dict = r.json()
    self.save_page(response_dict)

```

3. As the list is paginated, we will add the `save_page` method to the `Command` class to save a single page of tracks. This method takes the top dictionary of a page as a parameter:

```

def save_page(self, d):
    for track_dict in d["toptracks"]["track"]:
        track = Track()
        track.name = force_unicode(track_dict["name"])
        track.artist = force_unicode(
            track_dict["artist"]['name'])
        track.url = force_unicode(track_dict["url"])
        track.save()
        image_dict = track_dict.get("image", None)
        if image_dict:
            image_url = image_dict[1]["#text"]
            image_response = requests.get(image_url)
            track.image.save(
                os.path.basename(image_url),
                File(StringIO(image_response.content))
            )
    if self.verbosity >= NORMAL:
        print smart_str(" - %s - %s" % \
            (track.artist, track.name))

```

4. To run the import, call this in the command line:

```
(myproject_env)$ python manage.py import_music_from_lastfm_as_json
```

How it works...

By using the `requests.get` method, we read the data from Last.fm passing query parameters as `params`. The response object has a built-in method called `json`, which converts a JSON string and returns a parsed dictionary.

We read the total pages value from that dictionary and then save the first page of results. Then, we get the second and later pages one by one and save them. One interesting part in the import is downloading and saving the image. Here, we also use `request.get` to retrieve the image data, then we pass it to `File` through `StringIO`, which is accordingly used in the `image.save` method. The first parameter of `image.save` is a filename that will be overwritten anyway by the value from the `upload_to` function and is necessary only for the file extension.

See also

- The *Importing data from an external XML* file recipe

Importing data from an external XML file

Last.fm also allows you to grab data from their services in XML format. In this recipe, I will show you how to do this.

Getting ready

To prepare to import top tracks from Last.fm in XML format, follow these steps:

1. Start with the first three steps from the Getting ready section in *Importing data from an external JSON file* recipe.
2. Then, let's check out what the structure of the XML endpoint is (http://ws.audioscrobbler.com/2.0/?method=tag.gettoptracks&tag=disco&api_key=xxx):

```
<lfm status="ok">
  <toptracks tag="Disco" page="1" perPage="50" totalPages="20"
total="1000">
    <track rank="2">
      <name>Get Lucky (feat. Pharrell Williams)</name>
      <duration>102</duration>
      <mbid/>
```

```

        <url>
            http://www.last.fm/music/Daft+Punk/_/Get+Lucky+(fe
at.+Pharrell+Williams)
        </url>
        <streamable fulltrack="0">0</streamable>
        <artist>
            <name>Daft Punk</name>
            <mbid>056e4f3e-d505-4dad-8ec1-d04f521cbb56</mbid>
            <url>http://www.last.fm/music/Daft+Punk</url>
        </artist>
        <image size="small">http://userserve-ak.last.fm/
serve/34s/92459761.png</image>
        <image size="medium">http://userserve-ak.last.fm/
serve/64s/92459761.png</image>
        <image size="large">http://userserve-ak.last.fm/
serve/126/92459761.png</image>
        <image size="extralarge">
            http://userserve-ak.last.fm/serve/300x300/92459761.png
        </image>
    </track>
    ...
</toptracks>
</lfm>

```

How to do it...

Execute these steps one by one to import the top tracks from Last.fm in XML format:

1. Create a `music` app with a `Track` model like in the previous recipe, if you've not already done this.
2. Then, create a management command, `import_music_from_lastfm_as_xml.py`. We will be using the `ElementTree` XML API that comes with Python to parse XML nodes, as follows:

```

#music/management/commands/import_music_from_lastfm_as_xml.py
# -*- coding: UTF-8 -*-
import os
import requests
from xml.etree import ElementTree
from StringIO import StringIO
from django.core.management.base import BaseCommand
from django.core.management.base import CommandError
from django.utils.encoding import smart_str, force_unicode
from django.conf import settings
from django.core.files import File

```

```
from music.models import Track

SILENT, NORMAL, VERBOSE, VERY_VERBOSE = 0, 1, 2, 3

class Command(BaseCommand):
    help = "Imports top tracks from last.fm as XML."

    def handle(self, *args, **options):
        self.verbosity = int(options.get("verbosity",
                                          NORMAL))

        r = requests.get(
            "http://ws.audioscrobbler.com/2.0/",
            params={
                "method": "tag.gettoptracks",
                "tag": "disco",
                "api_key": settings.LAST_FM_API_KEY,
            }
        )

        root = ElementTree.fromstring(r.content)
        total_pages = int(root.find("toptracks").\
                           attrib["totalPages"])

        if self.verbosity >= NORMAL:
            print "=== Tracks imported ==="

        self.save_page(root)
        for page_number in xrange(2, total_pages + 1):
            r = requests.get(
                "http://ws.audioscrobbler.com/2.0/",
                params={
                    "method": "tag.gettoptracks",
                    "tag": "disco",
                    "api_key": settings.LAST_FM_API_KEY,
                    "page": page_number,
                }
            )
            root = ElementTree.fromstring(r.content)
            self.save_page(root)
```

3. As the list is paginated, we will add a `save_page` method to the `Command` class to save a single page of tracks. This method takes the root node of the XML as a parameter, as follows:

```
def save_page(self, root):
    for track_node in root.findall("toptracks/track"):
        track = Track()
        track.name = force_unicode(
            track_node.find("name").text)
        track.artist = force_unicode(
            track_node.find("artist/name").text)
        track.url = force_unicode(
            track_node.find("url").text)
        track.save()
        image_node = \
            track_node.find("image[@size='medium']")
        if image_node is not None:
            image_response = \
                requests.get(image_node.text)
            track.image.save(
                os.path.basename(image_node.text),
                File(StringIO(image_response.content))
            )
        if self.verbosity >= NORMAL:
            print smart_str(" - %s - %s" % \
                (track.artist, track.name))
```

4. To run the import, call this in the command line:

```
(myproject_env)$ python manage.py import_music_from_lastfm_as_xml
```

How it works...

The process is analogous to the JSON approach. By using the `requests.get` method, we read the data from Last.fm passing query parameters as `params`. The content of the response is passed to the `ElementTree` parser and the `root` node is returned.

The `ElementTree` nodes have the `find` and `findall` methods where you can pass XPath queries to filter out specific sub-nodes.

This is a table of the available XPath syntax supported by `ElementTree`:

XPath Syntax Component	Meaning
<code>tag</code>	Selects all child elements with the given tag.
<code>*</code>	Selects all child elements.
<code>.</code>	Selects the current node.
<code>//</code>	Selects all sub-elements on all levels beneath the current element.
<code>..</code>	Selects the parent element.
<code>[@attrib]</code>	Selects all elements that have the given attribute.
<code>[@attrib='value']</code>	Selects all elements for which the given attribute has the given value.
<code>[tag]</code>	Selects all elements that have a child named <code>tag</code> . Only immediate children are supported.
<code>[position]</code>	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression <code>last()</code> (for the last position), or a position relative to the last position (for example, <code>last()-1</code>).

So by using `root.find("toptracks").attrib["totalPages"]`, we read the total amount of pages. We save the first page and then go through the other pages one by one and save them too.

In the `save_page` method, `root.findall("toptracks/track")` returns an iterator through the `<track>` nodes under the `<toptracks>` node. With `track_node.find("image[@size='medium']")`, we get the medium-sized image.

See also

- The *Importing data from an external JSON file* recipe

Creating filterable RSS feeds

Django comes with a syndication feed framework that allows you to create RSS and Atom feeds easily. RSS and Atom feeds are XML documents with specific semantics. They can be subscribed in an RSS reader such as Feedly, or they can be aggregated into other websites, mobile applications, or desktop applications. In this recipe, we will create `BulletinFeed`, which provides a bulletin board with images. Moreover, the results will be filterable by URL query parameters.

Getting ready

Create a new `bulletin_board` app and put it under `INSTALLED_APPS` in the settings.

How to do it...

We will create a `Bulletin` model and an RSS feed for it that can be filtered by type or category, so that the visitor can subscribe only to bulletins that are, for example, offering used books:

1. In the `models.py` file of that app, add the `models` `Category` and `Bulletin` with a foreign key relationship between them:

```
#bulletin_board/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from django.core.urlresolvers import reverse
from utils.models import CreationModificationDateMixin
from utils.models import UrlMixin

TYPE_CHOICES = (
    ("searching", _("Searching")),
    ("offering", _("Offering")),
)

class Category(models.Model):
    title = models.CharField(_("Title"), max_length=200)

    def __unicode__(self):
        return self.title

    class Meta:
        verbose_name = _("Category")
        verbose_name_plural = _("Categories")
```

```
class Bulletin(CreationModificationDateMixin, UrlMixin):
    bulletin_type = models.CharField(_("Type"),
                                     max_length=20, choices=TYPE_CHOICES)
    category = models.ForeignKey(Category,
                                 verbose_name=_("Category"))
    title = models.CharField(_("Title"), max_length=255)
    description = models.TextField(_("Description"),
                                   max_length=300)
    contact_person = models.CharField(_("Contact person"),
                                      max_length=255)
    phone = models.CharField(_("Phone"), max_length=200,
                             blank=True)
    email = models.CharField(_("Email"), max_length=254,
                             blank=True)
    image = models.ImageField(_("Image"), max_length=255,
                              upload_to="bulletin_board/", blank=True)

    class Meta:
        verbose_name = _("Bulletin")
        verbose_name_plural = _("Bulletins")
        ordering = ("-created",)

    def __unicode__(self):
        return self.title

    def get_url_path(self):
        return reverse("bulletin_detail", kwargs={"pk": self.pk})
```

2. Then, create `BulletinFilterForm` that allows the visitor to filter bulletins by type and by category, as follows:

```
#bulletin_board/forms.py
# -*- coding: UTF-8 -*-
from django import forms
from django.utils.translation import ugettext_lazy as _
from models import Category, TYPE_CHOICES

class BulletinFilterForm(forms.Form):
    bulletin_type = forms.ChoiceField(
        label=_("Bulletin Type"),
        required=False,
        choices=("", "-----"), + TYPE_CHOICES,
    )
    category = forms.ModelChoiceField(
        label=_("Category"),
```

```

        required=False,
        queryset=Category.objects.all(),
    )

```

3. Add a `feeds.py` file with the `BulletinFeed` class inside, as follows:

```

#bulletin_board/feeds.py
# -*- coding: UTF-8 -*-
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse

from models import Bulletin, TYPE_CHOICES
from forms import BulletinFilterForm

class BulletinFeed(Feed):
    description_template = \
        "bulletin_board/feeds/bulletin_description.html"

    def get_object(self, request, *args, **kwargs):
        form = BulletinFilterForm(data=request.REQUEST)
        obj = {}
        if form.is_valid():
            obj = {
                "bulletin_type": \
                    form.cleaned_data["bulletin_type"],
                "category": form.cleaned_data["category"],
                "query_string": \
                    request.META["QUERY_STRING"],
            }
        return obj

    def title(self, obj):
        t = u"My Website - Bulletin Board"
        # add type "Searching" or "Offering"
        if obj.get("bulletin_type", False):
            tp = obj["bulletin_type"]
            t += u" - %s" % dict(TYPE_CHOICES)[tp]
        # add category
        if obj.get("category", False):
            t += u" - %s" % obj["category"].title
        return t

    def link(self, obj):
        if obj.get("query_string", False):

```

```

        return reverse("bulletin_list") + "?" + \
            obj["query_string"]
    return reverse("bulletin_list")

def feed_url(self, obj):
    if obj.get("query_string", False):
        return reverse("bulletin_rss") + "?" + \
            obj["query_string"]
    return reverse("bulletin_rss")

def item_pubdate(self, item):
    return item.created

def items(self, obj):
    qs = Bulletin.objects.order_by("-created")
    if obj.get("bulletin_type", False):
        qs = qs.filter(
            bulletin_type=obj["bulletin_type"],
        ).distinct()
    if obj.get("category", False):
        qs = qs.filter(
            category=obj["category"],
        ).distinct()
    return qs[:30]

```

4. Create a template for the bulletin description in the feed as follows:

```

{#templates/bulletin_board/feeds/bulletin_description.html#}
{% if obj.image %}
    <p><a href="{{ obj.get_url }}"></a></p>
{% endif %}
<p>{{ obj.description }}</p>

```

5. Create a URL configuration for the bulletin board app and include it in the root URL configuration, as follows:

```

#templates/bulletin_board/urls.py
# -*- coding: UTF-8 -*-
from django.conf.urls import *
from feeds import BulletinFeed

urlpatterns = patterns("bulletin_board.views",
    url(r"^$", "bulletin_list", name="bulletin_list"),
    url(r"^(?P<bulletin_id>[0-9]+)/$", "bulletin_detail",
        name="bulletin_detail"),
    url(r"^rss/$", BulletinFeed(), name="bulletin_rss"),
)

```

6. You will also need the views and templates for the filterable list and details of the bulletins. In the `Bulletin` list page template, add this link:

```
<a href="{% url 'bulletin_rss' %}"?{{ request.META.QUERY_STRING }}">RSS Feed</a>
```

How it works...

So, if you have some data in the database and you open `http://127.0.0.1:8000/bulletin-board/rss/?bulletin_type=offering&category=4` in your browser, you will get an RSS feed of bulletins with the type "Offering" and category ID 4.

The `BulletinFeed` class has the `get_objects` method that takes the current `HttpRequest` and defines the `obj` dictionary used in other methods of the same class. The `obj` dictionary contains the bulletin type, category, and current query string.

The `title` method returns the title of the feed. It can either be generic or related to the selected bulletin type or category. The `link` method returns the link to the original bulletin list with the filtering done. The `feed_url` method returns the URL of the current feed. The `items` method does the filtering itself and returns a filtered `QuerySet` of bulletins. And finally, the `item_pubdate` method returns the creation date of the bulletin.

To see all the available methods and properties of the `Feed` class that we are extending, refer to the following documentation:

<https://docs.djangoproject.com/en/1.6/ref/contrib/syndication/#feed-class-reference>

The other parts of the code are kind of self-explanatory.

See also

- ▶ The *Creating a model mixin with URL-related methods* recipe in *Chapter 2, Database Structure*
- ▶ The *Creating a model mixin to handle creation and modification dates* recipe in *Chapter 2, Database Structure*
- ▶ The *Using Tastypie to provide data to third parties* recipe

Using Tastypie to provide data to third parties

Tastypie is a web service API framework for Django models. It supports full GET/POST/PUT/DELETE/PATCH methods to deal with data. In this recipe, we will learn how to provide bulletins to third parties for reading.

Getting ready

First of all, install Tastypie to your virtual environment using the following command:

```
(myproject_env)$ pip install django-tastypie
```

Add tastypie to `INSTALLED_APPS` in the settings. Then, enhance the `bulletin_board` app that we defined in the *Creating filterable RSS feeds* recipe.

How to do it...

We will create an API for bulletins and inject it into the URL configuration as follows:

1. In the `bulletin_board` app, create an `api.py` file with two resources, `CategoryResource` and `BulletinResource`:

```
#bulletin_board/api.py
# -*- coding: UTF-8 -*-
from tastypie.resources import ModelResource
from tastypie.resources import ALL, ALL_WITH_RELATIONS
from tastypie.authentication import ApiKeyAuthentication
from tastypie.authorization import ReadOnlyAuthorization
from tastypie import fields
from models import Category, Bulletin

class CategoryResource(ModelResource):
    class Meta:
        queryset = Category.objects.all()
        resource_name = "categories"
        fields = ["title"]
        allowed_methods = ["get"]
        authentication = ApiKeyAuthentication()
        authorization = DjangoAuthorization()
        filtering = {
            "title": ALL,
        }
```

```

class BulletinResource(ModelResource):
    category = fields.ForeignKey(CategoryResource,
                                "category", full=True)

    class Meta:
        queryset = Bulletin.objects.all()
        resource_name = "bulletins"
        fields = ["bulletin_type", "category", "title",
                  "description", "contact_person", "phone",
                  "email", "image"]
        allowed_methods = ["get"]
        authentication = ApiKeyAuthentication()
        authorization = ReadOnlyAuthorization()
        filtering = {
            "bulletin_type": ALL,
            "title": ALL,
            "category": ALL_WITH_RELATIONS,
        }

```

2. Then, in the main URL configuration, include the API URLs as follows:

```

# -*- coding: UTF-8 -*-
from django.conf.urls import patterns, include, url
from django.conf import settings
from django.conf.urls.static import static
from django.contrib.staticfiles.urls import \
    staticfiles_urlpatterns

from django.contrib import admin
admin.autodiscover()

from tastypie.api import Api
from bulletin_board.api import CategoryResource, BulletinResource

v1_api = Api(api_name="v1")
v1_api.register(CategoryResource())
v1_api.register(BulletinResource())

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^api/', include(v1_api.urls)),
)

urlpatterns += staticfiles_urlpatterns()
urlpatterns += static(settings.MEDIA_URL,
    document_root=settings.MEDIA_ROOT)

```


3. Create a Tastypie API key for the `admin` user in the model administration. To do that, go to **Tastypie | Api key | Add Api key**, select the `admin` user, and save the entry. This will generate a random API key. Then, you can open this URL to see the JSON response in action (just replace `xxx` with your API key):

```
http://127.0.0.1:8000/api/v1/bulletins/?format=json&username=admin&api_key=xxx.
```

How it works...

Each endpoint of Tastypie should have a class extending `ModelResource` defined. Just like in Django models, the configuration of the resource is set in the `Meta` class:

- ▶ The `queryset` parameter defines the `QuerySet` of objects to list out.
- ▶ The `resource_name` parameter defines the name of the URL endpoint.
- ▶ The `fields` parameter lists out the fields of the model that should be shown in the API.
- ▶ The `allowed_methods` parameter lists out request methods, such as `get`, `post`, `put`, `delete`, and `patch`.
- ▶ The `authentication` parameter defines how third parties can authenticate themselves when connecting to the API. The available options are `Authentication`, `BasicAuthentication`, `ApiKeyAuthentication`, `SessionAuthentication`, `DigestAuthentication`, `OAuthAuthentication`, `MultiAuthentication`, or your own custom authentication. In our case, we are using `ApiKeyAuthentication`, because we want each user to use `username` and `api_key`.
- ▶ The `authorization` parameter answers the authorization question: is permission granted for this user to take this action? The possible choices are `Authorization`, `ReadOnlyAuthorization`, `DjangoAuthorization`, or your own custom authorization. In our case, we are using `ReadOnlyAuthorization`, because we only want to allow read access to the users.
- ▶ The `filtering` parameter defines by which fields one can filter the lists in the URL query parameters, for example, with the current configuration, you can filter items by titles that contain the word `movie`: `http://127.0.0.1:8000/api/v1/bulletins/?format=json&username=admin&api_key=xxx&title__contains=movie`.

Also, there is a `category` foreign key defined in `BulletinResource` with the `full=True` argument, meaning that the full list of category fields will be shown in the bulletin resource instead of just an endpoint link.

Besides JSON, Tastypie allows you to use other formats such as XML, YAML, and bplist.

There is a lot more you can do with APIs using Tastypie. To find out more details, check the official documentation:

<http://django-tastypie.readthedocs.org/en/latest/>

See also

- ▶ The *Creating filterable RSS feeds* recipe

10

Bells and Whistles

In this chapter, we will cover the following recipes:

- ▶ Using the Django shell
- ▶ The monkey patching slugification function
- ▶ The monkey patching model administration
- ▶ Toggling Debug Toolbar
- ▶ Using ThreadLocalMiddleware
- ▶ Caching the method value
- ▶ Getting detailed error reporting via e-mail
- ▶ Deploying on Apache with mod_wsgi
- ▶ Creating and using the Fabric deployment script

Introduction

In this chapter, we will go through several other important bits and pieces that will help you understand and utilize Django even better. You will get an overview of how to use the Django shell to experiment with the code before writing it into files. You will be introduced to monkey patching, also known as guerrilla patching, which is a powerful feature of dynamical languages such as Python, but should be used with moderation. You will learn how to debug your code and check its performance. Lastly, you will be taught how to deploy your Django project on a dedicated server.

Using the Django shell

With the virtual environment activated and your project directory selected as the current directory, enter this command into your command-line tool:

```
(myproject_env)$ python manage shell
```

By executing the preceding command, you get into an interactive Python shell configured for your Django project, where you can play around with the code and inspect classes, try out methods, or execute scripts on the fly. In this recipe, we will go through the most important functions you need to know to work with the Django shell.

Getting ready

Optionally, you can install IPython or bpython using one of the following commands, which will highlight the syntax for the output of your Django shell:

```
(myproject_env)$ pip install ipython
```

```
(myproject_env)$ pip install bpython
```

How to do it...

You can learn the basics of using the Django shell by following these instructions:

1. Run the Django shell by typing this command:

```
(myproject_env)$ python manage shell
```

The prompt will change to `In [1] : or >>>` depending on whether you use IPython or not.

2. Now you can import classes, functions, or variables and play around with them. For example, to see the version of an installed module, you can import the module and then try to read its `__version__`, `VERSION`, or `version` variables:

```
>>> import MySQLdb
>>> MySQLdb.__version__
'1.2.3'
```

3. To get a comprehensive description of a module, class, function, method, keyword, or documentation topic, use the `help` function. You can either pass a string with the path to a specific entity, or the entity itself, as follows:

```
>>> help('django.forms')
```

This will open the help page for the `django.forms` module. Use the arrow keys to scroll the page up and down. Press `Q` to get back to the shell.

- This is an example of passing an entity to the `help` function. This will open a help page for the `ModelForm` class:

```
>>> from django.forms import ModelForm
>>> help(ModelForm)
```

- To quickly see what fields and values are available for a model instance, use the `__dict__` attribute. Also use the `pprint` function to get dictionaries printed in a more readable format (not just one long line):

```
>>> from pprint import pprint
>>> from django.contrib.contenttypes.models import ContentType
>>> pprint(ContentType.objects.all()[0].__dict__)
{'_state': <django.db.models.base.ModelState object at
0x10756d250>,
 'app_label': u'bulletin_board',
 'id': 11,
 'model': u'bulletin',
 'name': u'Bulletin'}
```

Note that by using this method, we don't get many-to-many relationships. But this might be enough for a quick overview of the fields and values.

- To get all the available properties and methods of an object, you can use the `dir` function, as follows:

```
>>> dir(ContentType())
['DoesNotExist', 'MultipleObjectsReturned', '__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__unicode__', '__weakref__', '_base_manager', '_default_manager', '_deferred', '_do_insert', '_do_update', '_get_FIELD_display', '_get_next_or_previous_by_FIELD', '_get_next_or_previous_in_order', '_get_pk_val', '_get_unique_checks', '_meta', '_perform_date_checks', '_perform_unique_checks', '_save_parents', '_save_table', '_set_pk_val', '_state', 'app_label', 'clean', 'clean_fields', 'content_type_set_for_comment', 'date_error_message', 'delete', 'full_clean', 'get_all_objects_for_this_type', 'get_object_for_this_type', 'id', 'logentry_set', 'model', 'model_class', 'name', 'natural_key', 'objects', 'permission_set', 'pk', 'prepare_database_save', 'save', 'save_base', 'serializable_value', 'unique_error_message', 'validate_unique']
```

7. The Django shell is useful for experimenting with `QuerySets` or regular expressions before putting them into your model methods, views, or management commands. For example, to check the e-mail-validation regular expression, you can type this into the Django shell:

```
>>> import re
>>> email_pattern = re.compile(r'^@]+@[^@]+\.[^@]+')
>>> email_pattern.match('aidas@bendoraitis.lt')
<_sre.SRE_Match object at 0x1075681d0>
```

8. To exit the Django shell, press `Ctrl + D` or type the following command:

```
>>> exit()
```

How it works...

The difference between a normal Python shell and the Django shell is that when you run the Django shell, `manage.py` sets the `DJANGO_SETTINGS_MODULE` environment variable to the project's settings path, and then Django gets set up. So, all database queries, templates, URL configuration, or anything else are handled within the context of your project.

See also

- ▶ The *The monkey patching slugification function* recipe
- ▶ The *The monkey patching model administration* recipe

The monkey patching slugification function

Monkey patching or **guerrilla patching** is a piece of code that extends or modifies another piece of code at runtime. It is not recommended to use monkey patching often, but sometimes it is the only possible way to fix a bug in third-party modules without creating a separate branch of the module, or to prepare unit tests for testing without using complex database manipulations. In this recipe, you will learn how to exchange the default `slugify` function with the third-party `awesome-slugify` module, which handles German, Greek, and Russian words smarter and also allows custom slugification for other languages. The `slugify` function is used to create a URL-friendly version of the object's title or the uploaded filename: it strips the leading and trailing whitespace, converts the text to lowercase, removes nonword characters, and converts spaces to hyphens.

Getting ready

To get started, have a look at the following steps:

1. Install `awesome-slugify` to your virtual environment as follows:

```
(myproject_env)$ pip install awesome-slugify
```
2. Create an app named `guerrilla_patches` in your project and put it under `INSTALLED_APPS` in the settings.

How to do it...

In the `models.py` file of the `guerrilla_patches` app, add the following content:

```
#guerrilla_patches/models.py
# -*- coding: UTF-8 -*-
from django.utils import text
from slugify import slugify_de as awesome_slugify
awesome_slugify.to_lower = True
text.slugify = awesome_slugify
```

How it works...

The default Django `slugify` function handles German diacritical symbols incorrectly. To see that for yourself, run this code in the Django shell without the monkey patching:

```
(myproject_env)$ python manage.py shell
>>> from django.utils.text import slugify
>>> slugify(u"Heizölrückstoßabdämpfung")
u'heizolruckstoabdampung'
```

That is wrong in German, because the letter `ß` is totally stripped out instead of substituting it with `ss`, and the letters `ä`, `ö`, and `ü` are changed to `a`, `o`, and `u`, whereas they should be substituted with `ae`, `oe`, and `ue`.

The monkey patching we did loads the `django.utils.text` module at initialization and assigns the callable instance of the `Slugify` class as the `slugify` function. Now, if you run the same code in the Django shell, you will get different but correct results:

```
(myproject_env)$ python manage.py shell
>>> from django.utils.text import slugify
>>> slugify(u"Heizölrückstoßabdämpfung")
u'heizoelrueckstossabdaempfung'
```


To read more about how to utilize the `awesome-slugify` module, refer to:
<https://pypi.python.org/pypi/awesome-slugify>

See also

- ▶ The *Using the Django shell* recipe
- ▶ The *The monkey patching model administration* recipe

The monkey patching model administration

If you use multilingual fields, which we defined in *Chapter 2, Database Structures*, you might want to display the admin list view translated to the currently chosen language. When you set `title` to be `MultilingualCharField`, it creates the title as a virtual field (just like the `content_object` generic foreign key from a generic relationship), as well as the `title_*` fields of the `CharField` type for each language you support on the website. In this recipe, we will hack the default Django model administration to exchange virtual multilingual field names in `list_display` with the normal fields of the currently selected language. For example, `list_display = ["title", "author", "description"]` will become `list_display = ["title_lt", "author", "description_lt"]` when the Lithuanian language is selected as the current language for administration. This will serve as a more complex example of how monkey patching can be used.

Getting ready

Create an app named `guerrilla_patches` in your project (if you haven't done so before) and put it under `INSTALLED_APPS` in the settings.

How to do it...

Follow these steps to monkey patch the Django model administration and to see it in action:

1. In the `models.py` file for the `guerrilla_patches` app, insert this content:

```
#guerrilla_patches/models.py
# -*- coding: UTF-8 -*-
from django.contrib.admin.views import main
from django.utils.translation import get_language

class ExtendedChangeList(main.ChangeList):

    def get_translated_list(self, model, field_list):
        language = get_language()
```

```

        translated_list = []
        opts = model._meta
        virtual_fields = dict(
            (f.name, f) for f in opts.virtual_fields
        )
        for field_name in field_list:
            if field_name != "action_checkbox":
                field = virtual_fields.get(field_name, None)
                if field and field.__class__.\
                    __name__.startswith("Multilingual"):
                    field_name += "_" + language
                translated_list.append(field_name)
        return translated_list

    def __init__(self, request, model, list_display,
                  list_display_links, list_filter, date_hierarchy,
                  search_fields, list_select_related, list_per_page,
                  list_max_show_all, list_editable, model_admin):

        list_display = self.get_translated_list(model,
                                                  list_display)
        list_display_links = self.get_translated_list(model,
                                                         list_display_links)

        super(ExtendedChangeList, self).__init__(request,
                                                    model, list_display, list_display_links,
                                                    list_filter, date_hierarchy, search_fields,
                                                    list_select_related, list_per_page,
                                                    list_max_show_all, list_editable, model_admin)

main.ChangeList = ExtendedChangeList

```

2. Create an app named `books` that has a model, `Book`, with multilingual fields. Put this app under `INSTALLED_APPS` in the settings:

```

#books/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _
from utils.fields import MultilingualCharField
from utils.fields import MultilingualTextField

```

```
class Book(models.Model):
    author = models.CharField(_("author"), max_length=200)
    author = models.CharField(_("author"), max_length=200)
    title = MultilingualCharField(_("title"),
        max_length=200)
    description = MultilingualTextField(_("description"),
        blank=True)

    class Meta:
        verbose_name = _("book")
        verbose_name_plural = _("books")

    def __unicode__(self):
        return self.title
```

3. Add the Django model administration for this app, as follows:

```
#books/admin.py
# -*- coding: UTF-8 -*-
from django.contrib import admin
from django.conf import settings
from models import Book

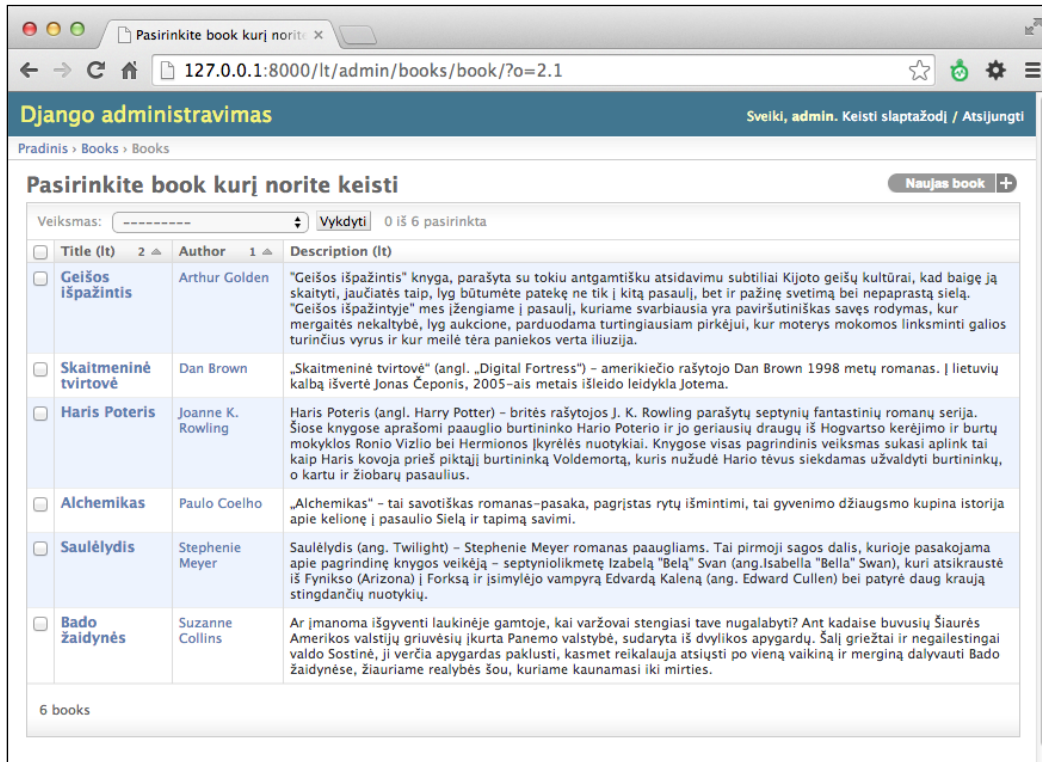
class BookAdmin(admin.ModelAdmin):
    list_display = ("title", "author", "description")
    list_display_links = ("title", "author")

    fields = ["author"] +\
        ["title_%s" % lang_code for
         lang_code, lang_name in settings.LANGUAGES] +\
        ["description_%s" % lang_code for
         lang_code, lang_name in settings.LANGUAGES]

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

How it works...

If you checkout the book list in the Django contributed model administration, you will see something like this:



Normally, virtual fields are not sortable. However, we have the fields of the selected language here (currently Lithuanian) and they are sortable in the graphical user interface of our Django model administration.

In the monkey patch, we extend the default Django administration's `ChangeList` class and modify the `list_display` and `list_display_links` attributes to return the field names of the specified language instead of the virtual fields.

The newly added method, `get_translated_list`, takes a model and the list of field names as parameters, checks the currently selected language, goes through the list of field names searching for virtual fields, and returns corresponding fields of the currently selected language if the virtual field class name starts with "Multilingual".

There's more...

Before creating a monkey patch, we need to completely understand how the code that we want to modify works. That can be done by analyzing existing code and inspecting the values of different variables. To do this, there is a useful built-in Python debugger module, `pdb`, which can temporarily be added to the Django code or any third-party module (using the following code) to stop the execution of a development server at any breakpoint:

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

This launches the interactive shell, where you can type in variables to see their values. If you type `c` or `continue`, the code execution will continue until the next breakpoint. If you type `q` or `quit`, the management command will be aborted. You can learn more commands of the Python debugger and how to inspect the traceback of the code at <https://docs.python.org/2/library/pdb.html>.

Another quick way to see a value of a variable in the development server is to raise a warning with the variable as a message, as follows:

```
raise Warning, some_variable
```

When you are in the `DEBUG` mode, the Django logger will provide you with the traceback and other local variables.



Don't forget to remove debugging functions before committing code to a repository.

See also

- ▶ The *Handling multilingual fields* recipe in *Chapter 2, Database Structures*
- ▶ The *Using the Django shell* recipe
- ▶ The *The monkey patching slugification function* recipe
- ▶ The *Toggling Debug Toolbar* recipe
- ▶ The *Getting detailed error reporting via e-mail* recipe

Toggling Debug Toolbar

While developing with Django, you will want to check the current template context, measure performance of SQL queries, or check the usage of caching. All that and more is possible with the Django Debug Toolbar. It is a configurable set of panels that display various debug information about the current request and response. In this recipe, I will guide you through how to toggle the visibility of the Debug Toolbar depending on a cookie, set by a bookmarklet. A bookmarklet is a bookmark of a small piece of JavaScript code that you can run on any page in a browser.

Getting ready

To get started with toggling the visibility of the Debug Toolbar, please have a look at the following steps:

1. Install the Django Debug Toolbar to your virtual environment:

```
(myproject_env)$ pip install django-debug-toolbar==1.2.1
```
2. Put `debug_toolbar` under `INSTALLED_APPS` in the settings.

How to do it...

Follow these steps to set up the Django Debug Toolbar, which can be switched on or off using bookmarklets in the browser:

1. Add these settings in your `settings.py` file:

```
#myproject/settings.py
MIDDLEWARE_CLASSES = (
    # ...
    "debug_toolbar.middleware.DebugToolbarMiddleware",
)

DEBUG_TOOLBAR_CONFIG = {
    "DISABLE_PANELS": [],
    "SHOW_TOOLBAR_CALLBACK": \
        "utils.misc.custom_show_toolbar",
    "SHOW_TEMPLATE_CONTEXT": True,
}

DEBUG_TOOLBAR_PANELS = [
    "debug_toolbar.panels.versions.VersionsPanel",
    "debug_toolbar.panels.timer.TimerPanel",
    "debug_toolbar.panels.settings.SettingsPanel",
    "debug_toolbar.panels.headers.HeadersPanel",
```

```
"debug_toolbar.panels.request.RequestPanel",
"debug_toolbar.panels.sql.SQLPanel",
"debug_toolbar.panels.templates.TemplatesPanel",
"debug_toolbar.panels.staticfiles.StaticFilesPanel",
"debug_toolbar.panels.cache.CachePanel",
"debug_toolbar.panels.signals.SignalsPanel",
"debug_toolbar.panels.logging.LoggingPanel",
"debug_toolbar.panels.redirects.RedirectsPanel",
]
```

2. In the `utils` module, create a file named `misc.py` with this function:

```
#utils/misc.py
# -*- coding: UTF-8 -*-
def custom_show_toolbar(request):

    return "1" == request.COOKIES.get("DebugToolbar", False)
```

3. Open the Chrome or Firefox browser and go to **Bookmark Manager**. Then, create two new JavaScript links there. The first link shows the toolbar. It looks like this:

- ❑ Name: Debug Toolbar On
- ❑ URL: `javascript:(function(){document.cookie="DebugToolbar=1; path=/";location.href=location.href;})();`

The second JavaScript link hides the toolbar and it looks like this:

- ❑ Name: Debug Toolbar Off
- ❑ URL: `javascript:(function(){document.cookie="DebugToolbar=0; path=/";location.href=location.href;})();`

How it works...

The `DEBUG_TOOLBAR_PANELS` setting defines the panels to be shown in the toolbar. The `DEBUG_TOOLBAR_CONFIG` setting dictionary defines the configuration for the toolbar including a path to the function, used to check whether to show the toolbar or not.

By default, when you browse through your project, the Django Debug Toolbar will not be shown. But as you click on your bookmarklet, **Debug Toolbar On**, the `DebugToolbar` cookie will be set to 1, the page will be refreshed, and you will see the toolbar with debugging panels. For example, you will be able to inspect the performance of SQL statements for optimization:

The screenshot shows the Django Debug Toolbar (DDT) interface. The main panel displays 'SQL queries from 1 connection' with a summary of 5 queries in 2.21 ms. Below this, a table lists the queries with their SQL text, execution time in milliseconds, and actions (Select, Explain).

Query	Timeline	Time (ms)	Action
QUERY = u'SELECT "django_session"."session_key", "django_session"."session_data", "django_session"."expire_date" FROM "django_session" WHERE ("django_session"."session_key" = %s AND "django_session"."expire_date" > %s)' - PARAMS = (u'ja7q6omy2mpysmzviodhr6irldjqzy52', u'2014-07-27 13:42:39.203685')	[Timeline bar]	0.86	[Sel] [Expl]
QUERY = u'SELECT "auth_user"."id", "auth_user"."password", "auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username", "auth_user"."first_name", "auth_user"."last_name", "auth_user"."email", "auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined" FROM "auth_user" WHERE "auth_user"."id" = %s' - PARAMS = (u'1')	[Timeline bar]	0.38	[Sel] [Expl]
QUERY = u'SELECT "auth_group"."id", "auth_group"."name" FROM "auth_group" - PARAMS = ()	[Timeline bar]	0.37	[Sel] [Expl]
QUERY = u'SELECT COUNT(DISTINCT "auth_user"."id") FROM "auth_user" - PARAMS = ()	[Timeline bar]	0.20	[Sel] [Expl]
QUERY = u'SELECT DISTINCT "auth_user"."id", "auth_user"."password", "auth_user"."last_login", "auth_user"."is_superuser", "auth_user"."username", "auth_user"."first_name", "auth_user"."last_name", "auth_user"."email", "auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined" FROM "auth_user" ORDER BY "auth_user"."username" ASC, "auth_user"."id" DESC' - PARAMS = ()	[Timeline bar]	0.41	[Sel] [Expl]

On the right side of the toolbar, there is a sidebar with various panels: Versions (Django 1.6.5), Time (CPU: 121.85ms (151.37ms)), Settings, Headers, Request (CHANGELIST_VIEW), SQL (5 queries in 2.21 ms), Templates (ADMIN/CHANGE_LIST.HTML), Static files (10 files used), Cache (0 calls in 0.00ms), Signals (8 receivers of 12 signals), Logging (0 messages), and Intercept redirects.

You will also be able to check the template context variables for the current view, as shown in the following screenshot:

The screenshot shows a web browser window with the URL `127.0.0.1:8000/admin/auth/user/`. The Django Debug Toolbar is visible at the bottom of the page. The 'Templates' panel is expanded, showing a list of templates rendered on the page. The first template is `admin/change_list.html`, located at `/Library/Python/2.7/site-packages/django/contrib/admin/templates/admin/change_list.html`. The second template is `admin/base_site.html`, located at `/Library/Python/2.7/site-packages/django/contrib/admin/templates/admin/base_site.html`. The third template is `admin/base.html`, located at `/Library/Python/2.7/site-packages/django/contrib/admin/templates/admin/base.html`. The context variables for the first template are displayed, including `'False': False`, `'None': None`, `'True': True`, `'action_form': <django.contrib.admin.helpers.ActionForm object at 0x111914390>`, `'actions_on_bottom': False`, `'actions_on_top': True`, `'actions_selection_counter': True`, `'app_label': u'auth'`, `'cl': <django.contrib.admin.views.main.Changelist object at 0x111908e50>`, `'has_add_permission': True`, `'is_popup': False`, `'media': <django.forms.widgets.Media object at 0x1119141d0>`, `'module_name': u'users'`, `'opts': <Options for User>`, `'preserved_filters': ''`, `'selection_note': u'0 of 1 selected'`, `'selection_note_all': u'1 selected'`, `'title': u'Select user to change'`, `{u'LANGUAGES': u'<languages>'}`, `u'LANGUAGE_BIDI': False`, `u'LANGUAGE_CODE': 'en-us'`, `u'MEDIA_URL': '/media/'`, `u'STATIC_URL': '/static/20140312022620/'`, `u'TIME_ZONE': 'UTC'`.

The right sidebar of the Django Debug Toolbar is visible, showing various panels: Versions (Django 1.6.5), Time (17 status, CPU: 121.85ms (151.37ms)), Settings (USER STATUS), Headers, Request (CHANGELIST_VIEW), SQL (5 QUERIES IN 2.21ms), Templates (ADMIN/CHANGE_LIST.HTML), Static files (10 FILES USED), Cache (0 CALLS IN 0.00ms), Signals (8 RECEIVERS OF 12 SIGNALS), Logging (0 MESSAGES), and Intercept redirects.

See also

- ▶ The *The monkey patching model administration* recipe

Using ThreadLocalMiddleware

The `HttpRequest` object contains useful information about the current user, language, server variables, cookies, session, and so on. As a matter of fact, `HttpRequest` is provided in the views and middlewares, and then you can pass it or its attribute values to forms, model methods, model managers, templates, and so on. To make life easier, you can use the `ThreadLocalMiddleware` middleware that stores the current `HttpRequest` object into the globally accessed Python thread, and so you can access it from model methods, forms, signal handlers, and any other place that didn't have direct access to the `HttpRequest` object before. In this recipe, we will define that middleware.

Getting ready

Create the `utils` app and put it under `INSTALLED_APPS` in the settings.

How to do it...

1. Add a file named `middleware.py` in the `utils` app with the following content:

```
#utils/middleware.py
# -*- coding: UTF-8 -*-
from threading import local
_thread_locals = local()

def get_current_request():
    """ returns the HttpRequest object for this thread """
    return getattr(_thread_locals, "request", None)

def get_current_user():
    """ returns the current user if it exists or None
        otherwise """
    request = get_current_request()
    if request:
        return getattr(request, "user", None)

class ThreadLocalMiddleware(object):
    """ Middleware that adds the HttpRequest object to
        thread local storage """
    def process_request(self, request):
        _thread_locals.request = request
```

2. Add this middleware to `MIDDLEWARE_CLASSES` in the settings:

```
#myproject/settings.py
MIDDLEWARE_CLASSES = (
    # ...
    "utils.middleware.ThreadLocalMiddleware",
)
```

How it works...

`ThreadLocalMiddleware` processes each request and stores the current `HttpRequest` object in the current thread. Each request-response cycle in Django is single-threaded. There are two functions: `get_current_request` and `get_current_user`. These functions can be used from anywhere to grab the current `HttpRequest` object or the current user.

For example, you can create and use `CreatorMixin`, which saves the current user as the creator of a model, as follows:

```
#utils/models.py
class CreatorMixin(models.Model):
    """
    Abstract base class with a creator
    """
    creator = models.ForeignKey(
        "auth.User",
        verbose_name=_("creator"),
        editable=False,
        blank=True,
        null=True,
    )

    def save(self, *args, **kwargs):
        from utils.middleware import get_current_user
        if not self.creator:
            self.creator = get_current_user()
        super(CreatorMixin, self).save(*args, **kwargs)
    save.alters_data = True

    class Meta:
        abstract = True
```

See also

- ▶ The *Creating a model mixin with URL-related methods* recipe in *Chapter 2, Database Structure*
- ▶ The *Creating a model mixin to handle creation and modification dates* recipe in *Chapter 2, Database Structure*
- ▶ The *Creating a model mixin to take care of meta tags* recipe in *Chapter 2, Database Structure*
- ▶ The *Creating a model mixin to handle generic relations* recipe in *Chapter 2, Database Structure*

Caching the method value

If you call the same model method with heavy calculations or database queries multiple times in the request-response cycle, the performance of the view might be very slow. In this recipe, you will learn about a pattern that you can use to cache the value of a method for later repetitive use. Note that we are not using the Django cache framework here, but just what Python provides us by default.

Getting ready

Choose an app with a model which has a time-consuming method that will be used repetitively in the same request-response cycle.

How to do it...

This is a pattern that you can use to cache a method value of a model for repetitive use in views, forms, or templates:

```
#someapp/models.py
# -*- coding: UTF-8 -*-
from django.db import models

class SomeModel(models.Model):
    # ...
    def some_expensive_function(self):
        if not hasattr(self, "_expensive_value_cached"):
            # do some heavy calculations...
            # ... and save the result to result variable
            self._expensive_value_cached = result
        return self._expensive_value_cached
```

How it works...

The method checks whether the `_expensive_value_cached` attribute exists for the model instance. If it doesn't exist, the time-consuming calculations are done and the result is assigned to this new attribute. At the end of the method, the cached value is returned. Of course, if you have several weighty methods, you will need to use different attribute names to save each calculated value.

You can now use something like `{{ object.some_expensive_function }}` in the header and the footer of a template and the time-consuming calculations will be done just once.

Or, you can use the function in the `if` condition to check whether the value exists, and also to print the value in the template:

```
{% if object.some_expensive_function %}
    <span class="special">{{ object.some_expensive_function }}</span>
{% endif %}
```

See also

- ▶ *Chapter 4, Templates and JavaScript*

Getting detailed error reporting via e-mail

To perform system logging, Django uses Python's built-in logging module. The default Django configuration seems to be quite complex. In this recipe, you will learn how to tweak it to send error e-mails with complete HTML, like what is provided by Django in the `DEBUG` mode when an error happens.

Getting ready

Locate the Django installation within your virtual environment.

How to do it...

The following procedure will help you send detailed e-mails about errors:

1. Open the `myproject_env/lib/python2.7/site-packages/django/utils/log.py` file in a text editor, and copy the `DEFAULT_LOGGING` dictionary to your project's settings as the `LOGGING` dictionary.

2. Add the "include_html": True configuration setting to the "mail_admins" handler, as follows:

```
#myproject/settings.py
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "filters": {
        "require_debug_false": {
            "()": "django.utils.log.RequireDebugFalse",
        },
        "require_debug_true": {
            "()": "django.utils.log.RequireDebugTrue",
        },
    },
    "handlers": {
        "console": {
            "level": "INFO",
            "filters": ["require_debug_true"],
            "class": "logging.StreamHandler",
        },
        "null": {
            "class": "django.utils.log.NullHandler",
        },
        "mail_admins": {
            "level": "ERROR",
            "filters": ["require_debug_false"],
            "class": "django.utils.log.AdminEmailHandler",
            "include_html": True,
        },
    },
    "loggers": {
        "django": {
            "handlers": ["console"],
        },
        "django.request": {
            "handlers": ["mail_admins"],
            "level": "ERROR",
            "propagate": False,
        },
        "django.security": {
            "handlers": ["mail_admins"],
            "level": "ERROR",
        },
    },
}
```

```
        "propagate": False,
    },
    "py.warnings": {
        "handlers": ["console"],
    },
}
}
```

How it works...

Logging configuration consists of four parts: loggers, handlers, filters, and formatters, described as follows:

- ▶ **Loggers** are entry points into the logging system. Each logger can have a log level: `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`. When a message is written to the logger, the log level of the message is compared with the logger's level. If it meets or exceeds the log level of the logger, it will be further processed by a handler. Otherwise, the message will be ignored.
- ▶ **Handlers** are engines that define what happens with each message in the logger. They can be written to a console, sent by an e-mail to the administrator, saved to a logfile, sent to the Sentry error logging service, and so on. In our case, we set the `include_html` parameter for the `mail_admins` handler as we want the full HTML with traceback and local variables for the error messages happening in our Django project.
- ▶ **Filters** provide additional control over the messages that are passed from loggers to handlers. For example, in our case, the e-mails will be sent only when the `DEBUG` mode is set to `False`.
- ▶ **Formatters** are used to define how to render a log message as a string. They are not used in this example, but for more information about logging, you can refer to the official documentation at <https://docs.djangoproject.com/en/1.6/topics/logging/>.

See also

- ▶ The *Deploying on Apache with mod_wsgi* recipe

Deploying on Apache with mod_wsgi

There are many options as to how to deploy your Django project. In this recipe, I will guide you through the deployment of a Django project on a dedicated Linux server with Virtualmin. A dedicated server is a type of Internet hosting where you lease the whole server not shared with anyone else. Virtualmin is a web hosting control panel that allows you to manage virtual domains, mailboxes, databases, and entire server without having deep knowledge of the command-line routines of server administration. You can try out Virtualmin at <http://www.virtualmin.com/demo>.

To run the Django project, we will be using the Apache web server with the `mod_wsgi` module and a MySQL database.

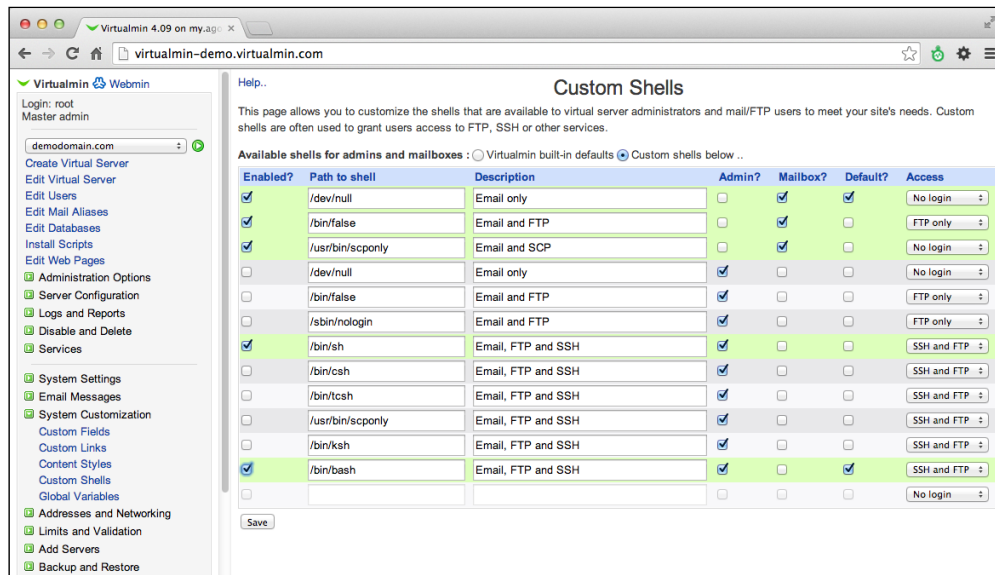
Getting ready

Make sure that you have Virtualmin installed on your dedicated Linux server. For instructions, refer to <http://www.virtualmin.com/download.html>.

How to do it...

Follow these steps to be able to deploy a Django project on a Linux server with Virtualmin:

1. Log in to Virtualmin as the root user and set `bash` instead of `sh` as the default shell for the server's users. That can be done by navigating to **Virtualmin | System Customization | Custom Shells**:



2. Create a virtual server for your project by navigating to **Virtualmin | Create Virtual Server**. Enable the following features: **Setup website for domain** and **Create MySQL database**. The username and password you set for the domain will also be used for SSH connections, FTP, and MySQL database access.

Virtualmin 4.09 on my.ag: x

virtualmin-demo.virtualmin.com

Virtualmin Webmin

Login: root
Master admin

demodomain.com

Create Virtual Server
Edit Virtual Server
Edit Users
Edit Mail Aliases
Edit Databases
Install Scripts
Edit Web Pages
Administration Options
Server Configuration
Logs and Reports
Disable and Delete
Services
System Settings
Email Messages
System Customization
Addresses and Networking
Limits and Validation
Add Servers
Backup and Restore
List Virtual Servers
System Statistics
System Information
Logout

Search:

Help..

Create Virtual Server

New virtual server type: Top-level server | Sub-server | Alias of demodomain.com | Alias of demodomain.com, with own email

New virtual server details

Domain name: myproject.com

Description:

Administration password:

Server configuration template: Default Settings

Account plan: Default Plan

Administration username: ☐ Automatic ☒ Custom username myproject

Advanced options

Enabled features

☐ Setup DNS zone? ☐ Accept mail for domain?

☒ Setup website for domain? ☐ Setup Webalizer for web logs?

☐ Setup SSL website too? ☒ Create MySQL database?

☐ Setup IP-based virtual FTP? ☐ Setup spam filtering?

☐ Setup virus filtering? ☐ Create Webmin login?

☐ Enable DAV logins? ☐ Enable AWstats reporting?

☐ Allow Mailman mailing lists?

IP address and forwarding

Initial website content

Create Server

3. Log in to your domain administration and set the **A** record for your domain to the IP address of your dedicated server.
4. Connect to the dedicated server via Secure Shell as the root user and install Python libraries, pip, virtualenv, MySQLdb, and Pillow system wide.
5. Ensure that the default MySQL database encoding is UTF-8:

```
$ ssh root@myproject.com
```

```
root@myproject.com's password:
```

```
$ nano /etc/mysql/my.cnf
```

Add the following command lines:

```
[client]
default-character-set=utf8

[mysql]
default-character-set=utf8

[mysqld]
collation-server = utf8_unicode_ci
init-connect='SET NAMES utf8'
character-set-server = utf8
```

Press *Ctrl* + *O* to save the changes and *Ctrl* + *X* to exit the nano editor.

Then, restart the MySQL server as follows:

```
$ /etc/init.d/mysql restart
```

Press *Ctrl* + *D* to exit Secure Shell.

6. Connect to the dedicated server via Secure Shell as a user of your Django project and create a virtual environment for your project, as follows:

```
$ ssh myproject@myproject.com
myproject@myproject.com's password:

$ virtualenv . --system-site-packages
$ echo source ~/bin/activate >> .bashrc
$ source ~/bin/activate
(myproject)myproject@server$
```

The `.bashrc` script will be called each time you connect to your Django project as a user via Secure Shell to the server. The `.bashrc` script will automatically set the virtual environment as active for this project.

7. If you host your project code on Bitbucket, you will need to set up SSH keys to avoid password prompts when pulling from or pushing to the Git repository. To do this, execute these commands one by one:

```
(myproject)myproject@server$ ssh-keygen
(myproject)myproject@server$ ssh-agent /bin/bash
(myproject)myproject@server$ ssh-add ~/.ssh/id_rsa
(myproject)myproject@server$ cat ~/.ssh/id_rsa.pub
```

This last command prints your SSH public key that you need to copy and paste at **Manage Account | SSH keys | Add Key** on the Bitbucket website.

8. Create a directory project, go into it, and clone your project's code as follows:

```
(myproject)myproject@server$ git clone git@bitbucket.org:somebitbucketuser/myproject.git myproject
```

Your project path should be something like this:

```
/home/myproject/project/myproject
```

9. Install the Python requirements for your project including a specified version of Django, as follows:

```
(myproject)myproject@server$ pip install -r requirements.txt
```

10. Create the media, tmp, and static directories under your project's directory. Also, create local_settings.py with settings like this:

```
#!/home/myproject/project/myproject/myproject/local_settings.py
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myproject",
        "USER": "myproject",
        "PASSWORD": "mypassword",
    }
}
PREPEND_WWW = True
DEBUG = False
ALLOWED_HOSTS = ["myproject.com"]
```

11. Import the database dump you created locally. If you are using a Mac, you can do that with an app, Sequel Pro, using an SSH connection. Or, you can upload the database dump to the server by FTP, and then run this in Secure Shell:

```
(myproject)myproject@server$ python manage.py dbshell < ~/db_backups/db.sql
```

12. Collect static files as follows:

```
(myproject)myproject@server$ python manage.py collectstatic --noinput
```

13. Go to the ~/public_html directory and create a wsgi file and .htaccess using the nano editor (or an editor of your choice). The .htaccess file will redirect all requests to your Django project, set in the wsgi file:

```
#!/home/myproject/public_html/myproject.wsgi
#!/home/myproject/bin/python
# -*- coding: utf-8 -*-
```

```

import os, sys, site
django_path = os.path.abspath(
    os.path.join(os.path.dirname(__file__),
        "../lib/python2.6/site-packages/"),
)
site.addsitedir(django_path)
project_path = os.path.abspath(
    os.path.join(os.path.dirname(__file__),
        "../project/myproject"),
)
sys.path += [project_path]
os.environ["DJANGO_SETTINGS_MODULE"] = "myproject.settings"
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()

# /home/myproject/public_html/.htaccess
AddHandler wsgi-script .wsgi
DirectoryIndex index.html
RewriteEngine On
RewriteBase /
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME}/index.html !-f
RewriteCond %{REQUEST_URI} !^/media
RewriteCond %{REQUEST_URI} !^/static
RewriteRule ^(.*)$ /myproject.wsgi/$1 [QSA,L]

```

14. Copy `.htaccess` as `.htaccess_live`.

15. Then, also create `.htaccess_under_construction` for maintenance cases. This new Apache configuration file will show `temporarily-offline.html` for all users except you, recognized by your IP address. You can check your IP by googling what 's my ip:

```

# /home/myproject/public_html/.htaccess_under_construction
AddHandler wsgi-script .wsgi
DirectoryIndex index.html
RewriteEngine On
RewriteBase /
RewriteCond %{REMOTE_HOST} !^93\.184\.216\.119
RewriteCond %{REQUEST_URI} !/temporarily-offline\.html$
RewriteCond %{REQUEST_URI} !^/media
RewriteCond %{REQUEST_URI} !^/static
RewriteRule .* /temporarily-offline.html [R=302,L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME}/index.html !-f
RewriteCond %{REQUEST_URI} !^/media
RewriteCond %{REQUEST_URI} !^/static
RewriteRule ^(.*)$ /myproject.wsgi/$1 [QSA,L]

```

Replace the IP digits in this file with your own IP.

16. Then, create an HTML file that will be shown when your website is down:

```
#/home/myproject/public_html/temporarily-offline.html
The site is being updated... Please come back later.
```

17. Log in to the server as the root user via Secure Shell and edit the Apache configuration as follows:

```
$ nano /etc/apache2/sites-available/myproject.mydomain.conf
```

Add these lines before `</VirtualHost>`:

```
Options -Indexes
AliasMatch ^/static/\d+/(.*) "/home/myproject/project/myproject/
static/$1"
AliasMatch ^/media/(.*) "/home/myproject/project/myproject/
media/$1"
<FilesMatch "\.(ico|pdf|flv|jpe?g|png|gif|js|css|swf)$">
    ExpiresActive On
    ExpiresDefault "access plus 1 year"
</FilesMatch>
```

Restart Apache for the changes to take effect:

```
/etc/init.d/apache2 restart
```

18. Set the default scheduled cron jobs. One of the ways to do that is to create small bash scripts for each management command that you want to execute regularly and then set them as scheduled tasks in Virtualmin. For example, to clean up your sessions every day, create a directory named `commands` and then a file named `cleanup.bsh` inside it:

```
#!/home/myproject/commands/cleanup.bsh
su myproject "-c source /home/myproject/bin/activate && cd /home/
myproject/project/myproject/ && python manage.py clearsessions"
```

Then, add an execution permission to this new file as follows:

```
(myproject)myproject@server$ chmod +x cleanup.bsh
```

Go to **Webmin | System | Scheduled Cron Jobs | Create a new scheduled cron job** and create a scheduled cron job with these properties:

- ☐ **Execute cron job as:** root
- ☐ **Command:** /home/myproject/commands/cleanup.bsh
- ☐ **Time and dates selected below:** Check
- ☐ **Minutes Selected:** 0
- ☐ **Hours Selected:** 3

Virtualmin 4.09 on my.ag: X
virtualmin-demo.virtualmin.com

Module Index

Create Cron Job

Job Details

Execute cron job as: root

Active? ☒ Yes ☐ No

Command: /home/myproject/commands/cleanup.bsh

Input to command:

Description:

When to execute

☐ Simple schedule .. ☒ Hourly ☐ Times and dates selected below ..

Minutes: ☒ All ☐ Selected ..

Hours: ☒ All ☐ Selected ..

Days: ☒ All ☐ Selected ..

Months: ☒ All ☐ Selected ..

Weekdays: ☒ All ☐ Selected ..

Note: Ctrl-click (or command-click on the Mac) to select and de-select minutes, hours, days and months.

Date range to execute

☒ Run on any date

☐ Only run from: Jan 1 to Jan 1

Create

Return to cron list

It will execute the cleanup of sessions every night at 3:00 am.

How it works...

With this configuration, files in media and static directories are served directly from Apache, whereas all other URLs are handled by the Django project through the `myproject.wsgi` file.

Using the `<FilesMatch>` directive in the Apache site configuration, all media files are set to be cached for one year. Static URL paths have a numbered prefix that changes whenever you update code from the Git repository.

When you need to update the website and want to set it down for maintenance, you'll have to copy `.htaccess_under_construction` to `.htaccess`. When you want to set the website up again, you'll have to copy `.htaccess_live` to `.htaccess`.

There's more...

To find other options for hosting your Django project, refer to:
<http://djangofriendly.com/hosts/>

See also

- ▶ The *Creating a project file structure* recipe in *Chapter 1, Getting Started with Django 1.6*
- ▶ The *Handling project dependencies with pip* recipe in *Chapter 1, Getting Started with Django 1.6*
- ▶ The *Setting up STATIC_URL dynamically for Git users* recipe in *Chapter 1, Getting Started with Django 1.6*
- ▶ The *Setting UTF-8 as the default encoding for MySQL configuration* recipe in *Chapter 1, Getting Started with Django 1.6*
- ▶ The *Creating and using the Fabric deployment script* recipe

Creating and using the Fabric deployment script

Usually, to update your site, you have to perform repetitive tasks, such as setting an under-construction page, stopping cron jobs, creating a database backup, pulling new code from a repository, migrating databases, collecting static files, testing, starting cron jobs again, and unsetting the under-construction page. That's quite tedious work, where mistakes can happen. Also, you need not forget different routines for staging site (the one where new features can be tested) and production site (which is shown to the public). Fortunately, there is a Python library called Fabric that allows you to automate these tasks. In this recipe, you will learn how to create `fabfile.py`, the script for Fabric, and how to deploy your project on staging and production environments.

The Fabric script can be called from the directory that contains it, as follows:

```
(myproject_env)$ fab staging deploy
```

This would deploy the project on the staging server.

Getting ready

Set up analogous staging and production websites using the instructions in the *Deploying on Apache with mod_wsgi* recipe. Install Fabric on your computer globally or into your project's virtual environment, as follows:

```
$ pip install fabric
```

How to do it...

In your project directory, create a `fabfile.py` file with several functions that will be called by script parameters:

```
#fabfile.py
# -*- coding: UTF-8 -*-
from fabric.api import env, run, prompt, local, get, sudo
from fabric.colors import red, green
from fabric.state import output

env.environment = ""
env.full = False
output['running'] = False

PRODUCTION_HOST = "myproject.com"
PRODUCTION_USER = "myproject"

def dev():
    """ chooses development environment """
    env.environment = "dev"
    env.hosts = [PRODUCTION_HOST]
    env.user = PRODUCTION_USER
    print("LOCAL DEVELOPMENT ENVIRONMENT\n")

def staging():
    """ chooses testing environment """
    env.environment = "staging"
    env.hosts = ["staging.myproject.com"]
    env.user = "myproject"
    print("STAGING WEBSITE\n")

def production():
    """ chooses production environment """
    env.environment = "production"
    env.hosts = [PRODUCTION_HOST]
    env.user = PRODUCTION_USER
    print("PRODUCTION WEBSITE\n")

def full():
    """ all commands should be executed without questioning """
    env.full = True

def deploy():
    """ updates the chosen environment """
    if not env.environment:
```



```
while env.environment not in ("dev", "staging",
                              "production"):
    env.environment = prompt(red('Please specify target '
                                'environment ("dev", "staging", or '
                                '"production"): '))
    print
globals()["_update_%s" % env.environment]()
```

The dev, staging, and production functions set the appropriate environment for the current task. Then, the deploy function calls the `_update_dev`, `_update_staging`, or `_update_production` private functions respectively. Let's define those private functions in the same file as follows:

- ▶ The function for deploying in the development environment will optionally do these tasks:
 - ❑ Update the local database with data from the production database
 - ❑ Download media files from the production server
 - ❑ Update code from the Git repository
 - ❑ Migrate the local database

Let's create this function in the Fabric script:

```
def _update_dev():
    """ updates development environment """
    run("") # password request
    print

    if env.full or "y" == prompt(red("Get latest "
                                     "production database (y/n)?"), default="y"):
        print(green(" * creating production-database "
                    "dump..."))
        run("cd ~/db_backups/ && ./backupdb.bsh --latest")
        print(green(" * downloading dump..."))
        get("~/db_backups/db_latest.sql",
            "tmp/db_latest.sql")
        print(green(" * importing the dump locally..."))
        local("python manage.py dbshell < "
            "tmp/db_latest.sql && rm tmp/db_latest.sql")
        print
        if env.full or "y" == prompt("Call prepare_dev "
                                     "command (y/n)?", default="y"):
            print(green(" * preparing data for "
                        "development..."))
            local("python manage.py prepare_dev")
    print
```

```

if env.full or "y" == prompt(red("Download media "
    "uploads (y/n)?"), default="y"):
    print(green(" * creating an archive of media "
        "uploads..."))
    run("cd ~/project/myproject/media/ "
        "&& tar -cz -f "
        "~/project/myproject/tmp/media.tar.gz *")
    print(green(" * downloading archive..."))
    get("~/project/myproject/tmp/media.tar.gz",
        "tmp/media.tar.gz")
    print(green(" * extracting and removing archive "
        "locally..."))
    for host in env.hosts:
        local("cd media/ "
            "&& tar -xzf ../tmp/media.tar.gz "
            "&& rm tmp/media.tar.gz")
    print(green(" * removing archive from the "
        "server..."))
    run("rm ~/project/myproject/tmp/media.tar.gz")
print

if env.full or "y" == prompt(red("Update code (y/n)?"),
    default="y"):
    print(green(" * updating code..."))
    local("git pull")
print

if env.full or "y" == prompt(red("Migrate database "
    "schema (y/n)?"), default="y"):
    print(green(" * migrating database schema..."))
    local("python manage.py migrate --no-initial-data")
    local("python manage.py syncdb")
print

```

- The function for deploying in a staging environment will optionally do these tasks:
 - ❑ Set a maintenance screen telling that the site is being updated and the visitors should wait or come later
 - ❑ Stop scheduled cron jobs
 - ❑ Get the latest data from the production database
 - ❑ Get the latest media files from the production database
 - ❑ Pull code from the Git repository
 - ❑ Collect static files
 - ❑ Migrate the database schema

- ❑ Restart the Apache web server
- ❑ Start scheduled cron jobs
- ❑ Unset the maintenance screen

Let's create this function in the Fabric script:

```
def _update_staging():
    """ updates testing environment """
    run("") # password request
    print

    if env.full or "y" == prompt(red("Set under- "
        "construction screen (y/n)?"), default="y"):
        print(green(" * Setting maintenance screen"))
        run("cd ~/public_html/ "
            "&& cp .htaccess_under_construction .htaccess")
    print

    if env.full or "y" == prompt(red("Stop cron jobs "
        "(y/n)?"), default="y"):
        print(green(" * Stopping cron jobs"))
        sudo("/etc/init.d/cron stop")
    print

    if env.full or "y" == prompt(red("Get latest "
        "production database (y/n)?"), default="y"):
        print(green(" * creating production-database "
            "dump..."))
        run("cd ~/db_backups/ && ./backupdb.bsh --latest")
        print(green(" * downloading dump..."))
        run("scp %(user)s@%(host)s:"
            "~/db_backups/db_latest.sql "
            "~/db_backups/db_latest.sql" % {
                "user": PRODUCTION_USER,
                "host": PRODUCTION_HOST,
            })
        print(green(" * importing the dump locally..."))
        run("cd ~/project/myproject/ && python manage.py "
            "dbshell < ~/db_backups/db_latest.sql")
    print
    if env.full or "y" == prompt(red("Call "
        "prepare_staging command (y/n)?"),
        default="y"):
```

```

        print(green(" * preparing data for "
            " testing..."))
        run("cd ~/project/myproject/ "
            "&& python manage.py prepare_staging")
    print
    if env.full or "y" == prompt(red("Get latest media "
        " (y/n)?"), default="y"):
        print(green(" * updating media..."))
        run("scp -r %(user)s@%(host)s:"
            "~/project/myproject/media/* "
            "~/project/myproject/media/" % {
                "user": PRODUCTION_USER,
                "host": PRODUCTION_HOST,
            })
    print

    if env.full or "y" == prompt(red("Update code (y/n)?"),
        default="y"):
        print(green(" * updating code..."))
        run("cd ~/project/myproject "
            "&& git pull")
    print

    if env.full or "y" == prompt(red("Collect static "
        "files (y/n)?"), default="y"):
        print(green(" * collecting static files..."))
        run("cd ~/project/myproject "
            "&& python manage.py collectstatic --noinput")
    print

    if env.full or "y" == prompt(red("Migrate database "
        " schema (y/n)?"), default="y"):
        print(green(" * migrating database schema..."))
        run("cd ~/project/myproject "
            "&& python manage.py migrate "
            "--no-initial-data")
        run("cd ~/project/myproject "
            "&& python manage.py syncdb")
    print

    if env.full or "y" == prompt(red("Restart webserver "
        " (y/n)?"), default="y"):
        print(green(" * Restarting Apache"))

```

```

        sudo("/etc/init.d/apache2 graceful")
    print

    if env.full or "y" == prompt(red("Start cron jobs "
        "(y/n)?"), default="y"):
        print(green(" * Starting cron jobs"))
        sudo("/etc/init.d/cron start")
    print

    if env.full or "y" == prompt(red("Unset under-
        "construction screen (y/n)?"), default="y"):
        print(green(" * Unsetting maintenance screen"))
        run("cd ~/public_html/ "
            "&& cp .htaccess_live .htaccess")
    print

```

- ▶ The function for deploying in a production environment will optionally do these tasks:
 - ❑ Set the maintenance screen telling that the site is being updated and the visitors should wait or come later
 - ❑ Stop scheduled cron jobs
 - ❑ Back up the database
 - ❑ Pull code from the Git repository
 - ❑ Collect static files
 - ❑ Migrate the database schema
 - ❑ Restart the Apache web server
 - ❑ Start scheduled cron jobs
 - ❑ Unset the maintenance screen

Let's create this function in the Fabric script:

```

def _update_production():
    """ updates production environment """
    if "y" != prompt(red("Are you sure you want to "
        "update " + red("production", bold=True) + \
        " website (y/n)?"), default="n"):
        return

    run("") # password request
    print

    if env.full or "y" == prompt(red("Set under-
        "construction screen (y/n)?"), default="y"):

```

```

        print(green(" * Setting maintenance screen"))
        run("cd ~/public_html/ "
            "&& cp .htaccess_under_construction .htaccess")
    print
    if env.full or "y" == prompt(red("Stop cron jobs "
        " (y/n)?"), default="y"):
        print(green(" * Stopping cron jobs"))
        sudo("/etc/init.d/cron stop")
    print

    if env.full or "y" == prompt(red("Backup database "
        " (y/n)?"), default="y"):
        print(green(" * creating a database dump..."))
        run("cd ~/db_backups/ "
            "&& ./backupdb.bsh")
    print

    if env.full or "y" == prompt(red("Update code (y/n)?"),
        default="y"):
        print(green(" * updating code..."))
        run("cd ~/project/myproject/ "
            "&& git pull")
    print

    if env.full or "y" == prompt(red("Collect static "
        "files (y/n)?"), default="y"):
        print(green(" * collecting static files..."))
        run("cd ~/project/myproject "
            "&& python manage.py collectstatic --noinput")
    print

    if env.full or "y" == prompt(red("Migrate database "
        "schema (y/n)?"), default="y"):
        print(green(" * migrating database schema..."))
        run("cd ~/project/myproject "
            "&& python manage.py migrate "
            "--no-initial-data")
        run("cd ~/project/myproject "
            "&& python manage.py syncdb")
    print

    if env.full or "y" == prompt(red("Restart webserver "
        " (y/n)?"), default="y"):
        print(green(" * Restarting Apache"))

```

```
        sudo("/etc/init.d/apache2 graceful")
    print
    if env.full or "y" == prompt(red("Start cron jobs "
        "(y/n)?"), default="y"):
        print(green(" * Starting cron jobs"))
        sudo("/etc/init.d/cron start")
    print

    if env.full or "y" == prompt(red("Unset under-
        "construction screen (y/n)?"), default="y"):
        print(green(" * Unsetting maintenance screen"))
        run("cd ~/public_html/ "
            "&& cp .htaccess_live .htaccess")
    print
```

How it works...

Each non-private function in a `fabfile.py` file becomes a possible argument to be called from the command-line tool. To see all the available functions, run the following command:

```
(myproject_env)$ fab --list
```

Available commands:

deploy	updates the chosen environment
dev	chooses development environment
full	all commands should be executed without questioning
production	chooses production environment
staging	chooses testing environment

These functions are called in the same order as they are passed to the Fabric script, as follows:

- ▶ To deploy in a development environment, you would run:

```
(myproject_env)$ fab dev deploy
```

This will ask you questions like:

```
Get latest production database (y/n)? [y] _
```

When answered positively, a specific step will be executed.

- ▶ To deploy in a staging environment, you would run:

```
(myproject_env)$ fab staging deploy
```

- ▶ Finally, to deploy in a production environment, you would run:

```
(myproject_env)$ fab production deploy
```

For each step of deployment, you will be asked whether you want to do it or skip it. If you want to execute all the steps without any prompts (except password requests), add a `full` parameter to the deployment script like this:

```
(myproject_env)$ fab dev full deploy
```

The Fabric script utilizes several basic functions, as follows:

- ▶ `local`: This function is used to run a command locally in the current computer
- ▶ `run`: This function is used to run a command as a specified user on a remote server
- ▶ `prompt`: This function is used to ask a question
- ▶ `get`: This function is used to download a file from a remote server to a local computer
- ▶ `sudo`: This function is used to run a command as the root (or other) user

Fabric uses the Secure Shell connection to perform tasks on remote servers. Each `run` or `sudo` command is executed as a separate connection, so when you want to execute multiple commands at once, you have to either create a bash script on the server and call it from Fabric, or you have to separate the commands using the `&&` shell operator, which means *execute the next command only if the previous one was successful*.

We are also using the `scp` command to copy files from the production server to the staging server. The syntax of `scp` for recursively copying all files from a specified directory is like this:

```
scp -r myproject_user@myproject.com:/path/on/production/server/* \
/path/on/staging/server/
```

To make the output more user friendly, we are using colors, like this:

```
print(green(" * migrating database schema..."))
```

The deployment script expects you to have two management commands: `prepare_dev` and `prepare_staging`. It's up to you to decide what to put into those commands. Basically, you could change the super user password to a simpler one and change the site domain there. If you don't need such functionality, just remove that from the Fabric script.

The general rule of thumb is not to store any sensitive data in the Fabric script if it is saved in the Git repository. So, for example, to make a backup of the database, we are calling the `backupdb.bsh` script on the remote production server. The content of such a file could be something like this:

```
#~/db_backups/backupdb.bsh
#!/bin/bash
if [[ $1 = '--latest' ]]
then
    today="latest"
else
    today=$(date +%Y-%m-%d-%H%M)
```



```
fi
mysqldump --opt -u myproject -pmypassword myproject > db_$today.sql
```

You can make it executable with the following:

```
$ chmod +x backupdb.bsh
```

When the preceding command is run without parameters, it will create a database dump with the date and time in the filename, for example, `db_2014-04-24-1400.sql`:

```
$ ./backupdb
```

When the `--latest` parameter is passed, the filename of the dump will be `db_latest.sql`:

```
$ ./backupdb --latest
```

There's more...

Fabric scripts can be used not only for deployment, but for any routine that you need to perform on remote servers, for example, collecting translatable strings when you are using the Rosetta tool for translating `*.po` files online, rebuilding search indexes when you are using Haystack for full-text searches, creating backups on demand, calling custom management commands, and so on.

To learn more about Fabric, refer to the following URL:

<http://docs.fabfile.org/en/1.9/>

See also

- The *Deploying on Apache with mod_wsgi* recipe

Index

Symbols

`<body>` section 96
`.gitignore` file 25
`<head>` section 96
`<pdf:nextpage>` tag 91
`<pdf:pagecount>` tag 91
`<pdf:pagenumber>` tag 91

A

action attribute 72
admin actions
 creating 151-154
administration settings
 exchanging, for external apps 158, 159
aggregation functions
 URL 157
Ajax
 images, uploading by 118-125
allowed_methods parameter 234
Apache
 deploying, with `mod_wsgi` 257-263
API key
 URL 217
app
 converting, to CMS app 177, 178
args parameter 215
ascending parameter 198
authentication parameter 234
authorization parameter 234
awesome-slugify module
 URL 242

B

base.html template
 arranging 94-96
Bootstrap
 about 67
 URL 67
BulletinFeed class 231

C

categories
 rendering, in template 204-206
category administration interface
 creating, with `django-mptt-admin` 200, 201
 creating, with
 `django-mptt-tree-editor` 202, 203
category, in forms
 selecting, single selection
 field used 206-208
change form
 map, inserting 160-168
change list filters
 developing 155-157
change list page
 columns, customizing 147-151
checkbox list
 used, for selecting multiple categories in
 forms 208-211
class-based views
 composing 82-85
CMS app
 app, converting to 177, 178
CMS_LANGUAGES setting 174

CMS page

new fields, adding to 187-193

CMS_PLACEHOLDER_CONF setting 184

CMS plugin

writing 181-186

columns

in change list view, customizing 147-151

commands, Python debugger

URL 246

Comma-separated values (CSV) 213

content, as template

parsing, template tag created to 141-143

continuous scroll

implementing 108-110

contribute_to_class() method 51

conventions

following, for tags 128, 129

following, for template filters 128, 129

CV model 86

D

data

importing, from external JSON file 217-222

importing, from external XML file 222-226

importing, from local CSV file 213-215

importing, from local Excel file 215-217

Debug Toolbar

toggling 247-250

DEBUG_TOOLBAR_CONFIG setting 248

DEBUG_TOOLBAR_PANELS setting 248

deploy function 266

Detroit Electric Car 154

dev function 266

Django 127

django-ajax-uploader 118

Django CMS

about 169

templates, creating for 170-173

URL 170

django-crispy-forms

about 67

form layout, creating with 67-72

Django documentation

URL 33, 54

django-mptt-admin

category administration interface,

creating with 200, 201

django-mptt-tree-editor

category administration interface,

creating with 202, 203

Django project

URL 263

Django shell

using 238-240

django-south app 52

Django template system

extending 128, 129

Document Type Definition 96

E

EditorialContent plugin

creating 182-184

e-mail

error report, obtaining via 254-256

error reporting

obtaining, via e-mail 254-256

Experience model 86

external apps

administration settings,

exchanging for 158, 159

external dependencies

including, in project 14-16

external JSON file

data, importing from 217-222

external XML file

data, importing from 222-226

F

Fabric

URL 274

Fabric deployment script

creating 264-274

using 264-274

Fabric script, functions

get function 273

local function 273

prompt function 273

run function 273

sudo function 273

- facets dictionary 78**
- Feed class**
 - URL 231
- feed_url method 231**
- Fieldset object 71**
- fields parameter 234**
- filterable RSS feeds**
 - creating 227-231
- filtering parameter 234**
- filters 256**
- first media object**
 - extracting, for template filter creation 131, 132
- foreign key**
 - changing, to many-to-many field 54-56
- form**
 - HttpRequest, passing to 58, 59
 - save method, utilizing of 60, 61
- formatters 256**
- form layout**
 - creating, with django-crispy-forms 67-72
- form_method property 72**

G

- generic relations**
 - model mixin, creating for 41-44
- get_absolute_url() method 33, 35**
- get function 273**
- get_media_svn_revision function 19**
- get method 85**
- get_nodes method 181**
- get_thumbnail_picture_url method 66**
- get_url() method 35**
- get_url_path() method 35**
- Git ignore file**
 - creating 25
- Git users**
 - STATIC_URL, setting up for 19, 20
- grid view 201**
- guerrilla patching 240**

H

- handlers 256**
- help parameter 215**

- hierarchical categories**
 - creating 197-199
- HTML5 data attributes**
 - using 100-103
- HttpRequest**
 - passing, to form 58, 59
- HttpRequest object 251**

I

- images**
 - uploading 62-66
 - uploading, by Ajax 118-125
- include_self parameter 198**
- Internet Movie Database**
 - URL 108
- items method 231**

J

- JavaScript settings**
 - creating 98, 99
 - including 97-99
- jScroll plugin**
 - URL 108

L

- layout property 69**
- level field 198**
- lft field 198**
- Like widget**
 - implementing 110-116
- limit_object_choices_to parameter 45**
- link method 231**
- local CSV file**
 - data, importing from 213-215
- local Excel file**
 - data, importing from 215-217
- local function 273**
- local settings**
 - creating 21, 22
 - including 21, 22
- loggers 256**
- logging**
 - URL 256

M

many-to-many field

foreign key, changing to 54-56

map

inserting, into change form 160-168

mark_safe function 132

Mercury Skate 154

meta tags

model mixin, creating for 38-40

method attribute 72

method value

caching 253

model creation

model mixin, creating for 36, 37

model mixin

creating, for handling creation 36, 37

creating, for handling generic relations 41-44

creating, for handling modification

dates 36, 37

creating, for meta tags 38-40

creating, with URL-related methods 33-35

using 32, 33

modification dates

model mixin, creating for 36, 37

Modified Preorder Tree

Traversal (MPTT) 195, 196

mod_wsgi module

Apache, deploying with 257-263

monkey patching 240

monkey patching model

administration 242-246

monkey patching slugification

function 240, 241

multilingual fields

advantages 45

handling 45-51

multiple categories, forms

selecting, checkbox list used 208-211

N

navigation

attaching 179-181

NavigationNode class 181

new fields

adding, to CMS page 187-193

O

obj dictionary 231

object details

opening, in pop up 104-108

object lists

filtering 72-78

object_relation_mixin_factory function 45

object_relation_mixin_factory mixin 44

on page load 166

order

importing, in Python files 26, 27

overwritable app settings

defining 28, 29

P

PageExtension class 191

page menu

structuring 174-176

paginated lists

managing 79-82

PDF documents

generating 85-91

pip

project dependencies, handling with 13, 14

URL 14

Pisa xhtml2pdf library 85

pop up

object details, opening in 104-108

pprint function 239

production function 266

project

external dependencies, including 14-16

project dependencies

handling, with pip 13, 14

project file structure

creating 10, 11

prompt function 273

Python compiled files

deleting 26

Python files

order, importing in 26, 27

Python Image Library 8

Python Package Index (PyPI) 14

Q

QueryDict objects

URL 145

queryset parameter 234

QuerySet, template

loading, template tag created for 137-141

R

relative paths

defining, in settings 17

render method 141, 143

REQUEST object 85

request query parameters

modifying, template tag created to 143, 144

requests.get method 222

resource_name parameter 234

right field 198

run function 273

Ryno 154

S

save method

utilizing, of form 60-62

save() method 37

scp command 273

selected dictionary 78

single selection field

used, for selecting category in
forms 206-208

slugify function 240

South migrations

using 52, 53

src attribute 91

staging function 266

STATIC_URL

setting up, for Git users 19, 20

setting up, for Subversion users 18, 19

Submit object 72

Subversion ignore property

setting 23, 24

Subversion users

STATIC_URL, setting up for 18, 19

sudo function 273

T

tags

conventions, following for 128, 129

Tastypie

used, for providing data to third
parties 232-234

template

categories, rendering in 204-206

creating, for Django CMS 170-173

including, for template tag creation 134-137

template filter

conventions, following for 128, 129

creating, for extracting first media

object 131, 132

creating, for humanizing URLs 133, 134

creating, for keeping track of date 129-131

template tag

creating, for including template 134-137

creating, for loading QuerySet 139-141

creating, for loading QuerySet in
template 137

creating, for modifying request query

parameters 143, 144

creating, for parsing content as

template 141-143

third parties

data providing, Tastypie used 232-234

ThreadLocalMiddleware

using 251, 252

title method 231

tree view 201

U

Uni-Form 67

UrlMixin class 35

URL-related methods

model mixin, creating with 33-35

URLs

humanizing, for creating template

filter 133, 134

UTF-8

setting, as default encoding 22, 23

UTF-8, default encoding

setting, for MySQL configuration 22, 23

V**Virtualenv 8****virtual environment**

working with 8, 9

Virtualmin

instructions, URL 257

URL 257

X

xlrd library 217

XML endpoint

URL 222

XPath syntax

. 226

.. 226

* 226

// 226

[@attrib] 226

[@attrib='value'] 226

[position] 226

[tag] 226

tag 226



Thank you for buying Web Development with Django Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book *"Mastering phpMyAdmin for Effective MySQL Management"* in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

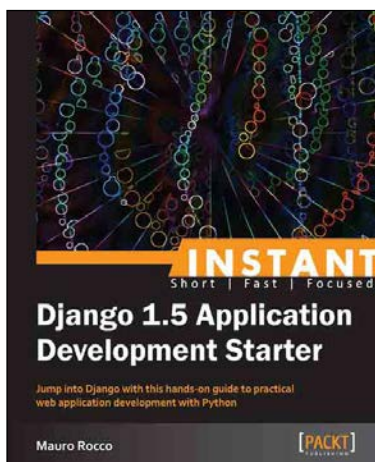
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



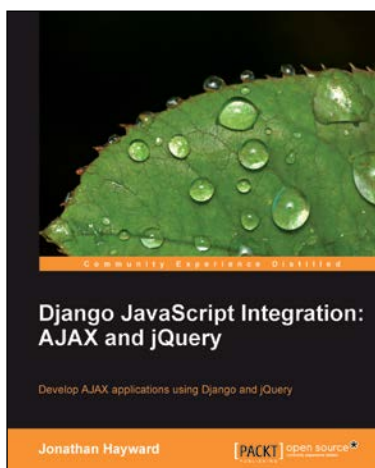
Instant Django 1.5 Application Development Starter

ISBN: 978-1-78216-356-5

Paperback: 78 pages

Jump into Django with this hands-on guide to practical web application development with Python

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Work with the database API to create a data-driven app.
3. Learn Django by creating a practical web application.
4. Get started with Django's powerful and flexible template system.



Django JavaScript Integration: AJAX and jQuery

ISBN: 978-1-84951-034-9

Paperback: 324 pages

Develop AJAX applications using Django and jQuery

1. Learn how Django + jQuery = AJAX.
2. Integrate your AJAX application with Django on the server side and jQuery on the client side.
3. Learn how to handle AJAX requests with jQuery.
4. Compare the pros and cons of client-side search with JavaScript and initializing a search on the server side via AJAX.

Please check www.PacktPub.com for information on our titles

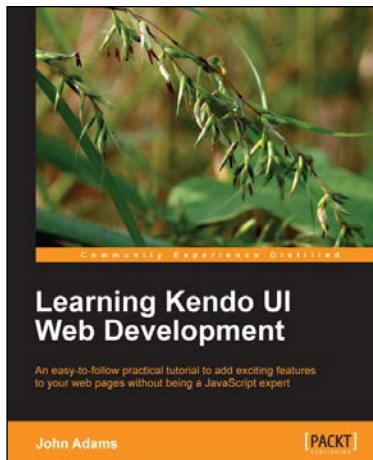


Web Development with Jade

ISBN: 978-1-78328-635-5 Paperback: 80 pages

Utilize the advanced features of Jade to create dynamic web pages and significantly decrease development time

1. Make your templates clean, beautiful, and reusable.
2. Use Jade best practices right from the start.
3. Successfully automate redundant markup.



Learning Kendo UI Web Development

ISBN: 978-1-84969-434-6 Paperback: 288 pages

An easy-to-follow practical tutorial to add exciting features to your web pages without being a JavaScript expert

1. Learn from clear and specific examples on how to utilize the full range of the Kendo UI tool set for the web.
2. Add powerful tools to your website supported by a familiar and trusted name in innovative technology.
3. Learn how to add amazing features with clear examples and make your website more interactive without being a JavaScript expert.

Please check www.PacktPub.com for information on our titles