VOLUME 10 | ISSUE 08

digit FastTrack

YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

To

BUILDING YOUR OWN CUSTOM CMS USING DJANGO

- Getting started
- Python basics
- Starting with DJango
- Designing your CMS
- Django models
- Django URL routing
- Django views
- Django templates
- Django admin
- Conclusion

A 9.9 Media Publication

www.digit.in/forum

Join the forum to express your views and resolve your differences in a more civilised way.

digit.in FORUM

Post your queries and get instant answers to all your technology related questions

One of the most active **online technology forums** not only in India but world-wide

JOIN NOW

digit.in

FAST TRACK to

BUILDING YOUR OWN CUSTOM CMS USING DJANGO

powered by

digit

YOUR TECHNOLOGY NAVIGATOR

CHAPTERS

BUILDING YOUR OWN CUSTOM CMS USING DJANGO
AUGUST 2015

CREDITS
The people behind this book

EDITORIAL

Executive Editor
Robert Sovereign-Smith

Associate Editor
Siddharth Parwatay

Writer
Kshitij Sobti

DESIGN

Sr. Creative Director
Jayan K Narayanan

Sr. Art Director
Anil VK

Associate Art Director
Anil T

Sr. Visualisers
Shigil Narayanan
Sristi Maurya

Visualiser
Baiju NV

Sr. Designer
Pradeep G Nair

digit.in

server

COVER DESIGN: ANIL T

Introduction

We know one size doesn't fit all, but do hundreds of sizes fit all? A question for philosophers, so we'll leave it for them to answer. What we're hinting at is that there are hundreds of CMS systems out there, but we know there is a CMS-shaped hole in your heart that can only be filled by something you create yourself. Well, we have you covered.

Creating a CMS may be the goal of this booklet, but as the saying goes, the journey is more important than the destination. In learning how to create a CMS we will explore the popular open source web framework Django, and Python, the language it is built on. Even if your heart is content with existing CMS systems, you should hopefully find these technologies interesting and applicable to your needs.

We pick the popular open source Django framework because it provides the right mixture of features, performance and ease for our case.

In this booklet, we start with the basics of Python, to accommodate those who have never programmed in it before, but some knowledge of programming, and web technologies (mostly HTML) is essential. Once you're familiar with Python, we'll move on to the Django framework. After introducing the framework we will then consider the design of our CMS, and how best to utilise Django in achieving that design. While developing each piece of functionality we will also take a deeper look into the features of Django that power them.

The best way to use this booklet is to follow along on your computer. If you find that some code is unclear, or does not work, chances are that the snippet of code you just typed in needs to be in the right context. We've made all code snippets available (in their correct context) online on GitHub for your reference. You can find the snippets by following this shortlink: *http://dgit.in/snipcoderep*

Likewise the complete project is also available online on GitHub. You can find it here: *http://dgit.in/finalcoderep*

This might seem counter-intuitive but it would be great if you don't follow instructions exactly, but instead try to add your own flavour to the code. In some places we will even mention things you can improve or change. If things break, you will learn more from knowing how and where than you will by just following instructions blindly.

With that we wish you all the best. Do let us know which topics you'd like us to cover in upcoming FastTracks by writing to *editor@digit.in*.

Happy reading.

GETTING STARTED

Do you have some content lying about? Do you regularly feel like it could use some management? Well, then a Content Management System is exactly what you need! And this chapter will introduce you to some of the basic concepts that'll put you well on your way to doing just that.

Chances are you have heard the term CMS or Content Management System before; it's quite possible in fact that you have used one as well. If you have used WordPress, or any blogging platform, you have used something close to (if not exactly categorised as) a CMS.

A CMS, or Content Management System, is, as you may have surmised from its name, a software package that you to manage 'content'. It allows users to add, edit, categorise and organise content, and possibly choose how it is displayed. The more interesting question perhaps is, what is content?

Well content is any piece of information / data / media / knowledge. To see what that boils down to let's look at WordPress. WordPress is a CMS written in the PHP programming language. It allows you to manage articles, categories, tags, media files, users, links and comments. If it only managed media files, we'd probably call it a gallery application; if it only managed users it would be a contact management application; If it only managed links you'd call it a bookmarking application. The fact that it manages all of these, and provides a way to customise how they are presented, and how they relate to each other is why people call it a CMS.

For instance, you can create multiple categories, and articles to these categories that have media files embedded in them, they have proper tags, and allow adding comments. Articles have an author, and you can see all articles by a particular author if you so wish. Users can have different roles. For instance while an administrator might be able to modify all aspects of the website, an author would only be able to add articles, and edit articles they themselves have created. An editor would be able to edit any article or create a new one, and a regular user can simply view the content.

WordPress is one of the most popular CMS software and for good reason. It is easy to install, set up, use and administer.

If you've always thought of WordPress as being a blogging platform, and are now confused by our instance that it is in fact a CMS, here is what you need to know. A blog is essentially a CMS that is specialised for regularly posting content, which is supposed to be displayed, accessed and consumed chronologically. CMSs that specialise in this use case are often called blogging platforms / frameworks / engines. While WordPress started with this specialisation it has since grown to encompass features that are useful for more general use cases beyond simply build a blog.

A website for a product, let's say a television set, just needs a collection of content that is categorised to provide different forms of information about the product. You might have the home page display enticing photos of the television set while a gallery page displays even more images. Another page could have video clips of the advertisements for that television set. A specifications page could have basic information about the product, while a support page could have a list of common issues, troubleshooting and warranty information. While this content might need to be updated from time to time, it will be laid out very differently than a blog, which would just have a series of entries over time.

Of course to manage such a site you will need the ability to create, store and organise text content, media content, and forms; all things that

WordPress can do. Our goal in this booklet will be to guide you to creating such a system.

CMS systems come in all shapes and sizes, designed for all kinds of use cases. Pick a language and chances are there is a CMS written in it — if that language is PHP there are dozens. There are open source CMSs such as WordPress, Drupal and Joomla, freeware ones like DotNetNuke, paid solutions like MovableType and even CMS SaaS options such as Contentful. They can range in price from free (Drupal) to (in technical parlance) a shit-tonne of money (Adobe CQ5).

Between all of these CMSs, most bases are covered. You can find one written in the language of your choice, whether that is PHP, Python, Ruby, JavaScript, Java or heck even Perl, Erlang and VB.NET. So why then should you make your own CMS in Django instead.

Well for one, because it's possible to do so. And secondly in doing so you can learn a lot about how Django works as well, and that is applicable to more than just CMSs. Also by making your own CMS focused on just the things you need, you can get an extremely light solution.

Since Django is Python-based, and we will need to be coding in Python as well, we'll first being with an introduction to Python. Feel free to skip this chapter if you are already familiar with the language.

PYTHON BASICS

Python is a beautiful, clear and easy to use language that you'll find very useful even when you're done with this booklet.

If you have never encountered Python before, you're in for a delightful surprise! Python is a rather popular programming language that you will find somewhere within the top 10 of any language popularity contest. It is famous for being quite readable, easy to understand, and easy to pick up. Hopefully you will find those adjectives appropriate once you are done with this chapter.

One of the unusual aspects of the Python syntax, that helps ensure that most Python code looks neat, clean, and uniform is that code formatting is (to an extent) part of the syntax of the language. Where many programming languages such as C / C++ / Java / C# / JavaScript use curly braces to demarcate blocks of code belonging within a function, class, loop or if statement, Python uses indentation.

If you use an IDE to write code in any of the languages we have mentioned above, chances are it will automatically indent code in addition to placing braces around to to make the code look neat and to give developers a better overview of the flow of the code. With Python there are no curly braces, the indentation itself enough — and why not, you would be indenting the code anyway!

Let's look at some simple code in JavaScript and Python to compare. The following code snippets implement one of the most important requirements of most programmers, the ability to distinguish large numbers from small numbers. Feel free to copy this code! You're welcome.

JavaScript Code:

```
function example(num) {
    if (num > 100) {
        return "Large Number";
    } else {
        return "Small Number";
    }
}
```

Equivalent Python Code:

```
def example(num):
    if num > 100:
        return "Large Number"
    else:
        return "Small Number"
```

Before you learn anything else about Python though, you need to know another thing that makes it a bit odd compared to other programming languages; there are two versions of Python available, and they both have a slightly different syntax. On the official Python website, and all across the internet you will see references to both Python 2 and Python 3.

To dispel any confusion as to which you should use, we'll just go ahead and recommend Python 3 at this point of time. It is the latest version of Python, and the future-proof one. The only reason Python 2 is still so popular is that Python 3 has a slightly different syntax that prevented people from quickly adopting it. This in turn meant that there was a larger code base, and a larger set of libraries available for Python 2 than for Python 3. And now nearly seven years after Python 3 was introduced that gap has reduced considerably.

If after learning Python 3 from this text you feel that you need to use Python 2 because some library or package that is not available for Python 3, worry not. Python 3 and 2 are mostly similar and while it may be hard to convert between them at times, learning one if you know the other isn't hard at all.

Before we dive into learning this great language, let's see how you can get it for the platform of your choice.

Getting and Installing Python

The most obvious place to get Python is the official Python website at *http://www.python.org*

As we have mentioned before, there are two versions of Python currently supported and available on the Python website. We will need to download and install the latest version of Python 3. As of this writing that is Python 3.4.3, although 3.5 is just a few months away.

There are also a number of non-official alternative Python editions available. Some of them are just versions that a commercially supported, however others bring their own unique features. For instance Jython is Python running on the JVM and as such can interface easily with Java code; IronPython runs *on .NET* and can *use .NET* features. Another interesting case is PyPy, which can be much faster than regular Python in many use cases. Enthought's Canopy is a good choice for beginners and comes with a simple IDE, and Anaconda Python is optimised for working with large datasets in scientific computing.

We'll just use the plain old official Python version here and you should too (although you can ActiveState's ActivePython since it is entirely compatible). When installing the official version of Python 3 on Windows make sure to select the option to add python.exe to the PATH environment variable since this makes Python programs much easier to run. Once it is installed, you can run Python scripts using the 'python' command. If you don't add python.exe to the PATH environment variable you will need to run Python scripts by providing the complete path as 'C:\Python34\python.exe' or wherever you have installed Python 3.

For Windows users what we have described above works well, however Linux, and OSX both come with Python pre-installed, and chances are it might not be Python 3. If you are using some Linux distribution, you may need to check if Python 3 is installed, and install it if it isn't. Here are the commands to install Python 3 on some of the popular distros:

Debian-based (already installed on Ubuntu): apt-get install python3

OpenSUSE-based: zypper install python3

RedHat-based (Fedora): yum install python3

Arch-based: already installed

For our readers running OSX on their Apple computers, you will find that Python 2 comes preinstalled with the OS. You can download an install

Python 3 from the Python website. Once you do so you should be able to run Python 3 scripts using the 'python3' command. The 'python' command will still be reserved for the default Python 2 install.

On most Linux distros Python 3 is only available via the 'python3' command while Python 2 is available via the 'python' or 'python2' commands.

On Windows and OSX installing Python also gives you access to a simple Python IDE called IDLE. You may wish to use this to write your code. If you are looking for a more powerful IDE you should check out PyCharm. It is available in a free and open source community edition that should suit our needs perfectly. There are a number of free IDEs available for Python. There are even a plugins available for Visual Studio and Eclipse if you use one of those.

Basc Syntax

Now that we have Python 3 installed there are two ways we can run Python code. The first is to start the Python interpreter and type commands into it to run them immediately, or you can type all your code into a text file, give it the '.py' extension and then run it with the 'python3' or 'python' command. The content of this section is simple enough for you to just use the interpreter.

As is customary let's start with a simple 'Hello World!' program:

- `print('Hello World!')`

Comments in Python are added using the '#' character. Anything following a '#' character in a line is treated as a comment and not executed.

Variables

In Python creating variables is as simple as giving them a value. So the following are all valid examples of creating variables:

- `my _ name = 'Kabir Sharma'`
- `my _ age = 11`
- `value _ of _ pi = 3.1412`

In Python the convention is to use Snake Case, i.e. to use lower case names for variables and functions and separate words using underscores. For constants the convention is to use upper case names separated by underscores.

Mathematical Operations

A common requirement is to perform mathematical operations, and Python is actually quite powerful at this. Python can work with arbitrarily large numbers, so if you were just looking for a calculator to calculate 2^2000, Python should have the answer for you (all 604 digits of it) beforekwin you can count to, well one.

All common operation such as addition, subtraction, multiplication and division work as you'd expect using their usual symbols (+, -, *, /). The modulo operator also uses the conventional '%' character. Exponentiation is performed using two asterisks, as: 2**2000

On difference between Python 2 and 3 here is that Python 3 will output 3/2 as 1.5 while Python 2 will floor that to 1.

```
>>> 2**2000
[illegible]
>>>
```

Complex mathematical operations are a cakewalk with Python

String Operations

Web application especially need to be able to handle string manipulations. Here too you will find Python's syntax quite beneficial.

Like with most programming languages, Python strings are surrounded by quotes. If you need to use a single quote in your string, use double quotes to encapsulate it and vice versa. If you need both, you can escape the quote character using a backslash, as you can see in the following examples of Python strings:

- `"A simple string"`
- `'There are no strings on me'`
- `` `"Of course" said noone in particular' ``
- `"Here's a string"`
- `'"Here\'s a string" said the previous line'`

Like with many languages, you can use '\t' for tabs and '\n' for the newline character.

Another convenience Python provides when dealing with long strings, is the triple quote. If you want to enter a para of text with all kinds of quotation marks and new lines, rather than littering it with escape characters you can use the triple quote as follows:

- `some _ html = """`

```
• <html>
• <head>
• <title>Some HTML in a Python string</title>
• <body>
• "We can use the quotation characters in here without
issues," the author exclaimed.
• Even this one: '
• </body>
• </html>
• """
```

There are also other conveniences afforded by Python for strings. Such as, concatenating two strings using the plus sign, repeating a string multiple times using the multiplication sign (*). Here are a few example:

```
• "You should never " + "ever " * 3 + "divide strings"
```

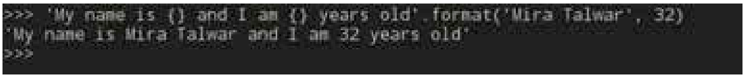
In Python as with many other languages, strings are arrays of characters, and as such you can use indexing to get to particular characters within the sequence. Python goes a bit further and allows for negative indexing, access sequences of characters within a string and more. Here are a few examples along with their outputs:

```
• a _ string = "FastTrack"
• print(a _ string[0])      # F
• print(a _ string[0:4])    # Fast
• print(a _ string[:4])     # Fast
• print(a _ string[4:])     # Track
• print(a _ string[-1])     # k
• print(a _ string[::2])    # FsTak
• print(a _ string[::-1])   # kcarTtsaF
```

Here's an overview, to get to a subsequence you can use the string[start:stop] syntax; if you omit the start it starts from the beginning, and if you omit the stop it goes till the end. Negative indexing counts characters from the end of the string rather than the beginning. You can provide a stride in the string[start:stop:stride] syntax to skip characters; so in a_string[::2] we are selecting every second character starting from the beginning and going to the end. In the final example where we provide a negative stride, it will iterate over the string backwards, thus giving us a reversed string in a single line of code.

Another final bit of string knowledge that can come in handy while building websites is string formatting. Here is how to accomplish that in Python:

```
• 'My name is {} and I am {} years old'.format('Mira Talwar',
32)
```

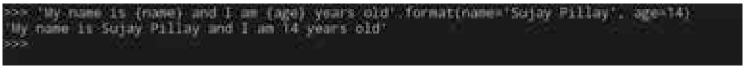


```
>>> 'My name is {} and I am {} years old'.format('Mira Talwar', 32)
'My name is Mira Talwar and I am 32 years old'
>>>
```

python-strings1

Here the '{}' bits are replaced by the parameters provided to the format method. You can make this a little more explicit as follows:

```
• 'My name is {name} and I am {age} years old'.
format(name='Sujay Pillay', age=14)
```



```
>>> 'My name is {name} and I am {age} years old'.format(name='Sujay Pillay', age=14)
'My name is Sujay Pillay and I am 14 years old'
>>>
```

python-strings2

Control Flow Statements

if statement

The Python if statement is quite uncomplicated as you can see from this perfectly silly example:

```
• if something < 11:
•     print(something)
• elif something > 22:
•     print(something * 2)
• else:
•     print(something * 3)
```

Python tries to deliberately keep the syntax minimal rather than supporting multiple ways of doing the same thing. For instance, there is no switch case syntax in Python because it is perfectly possible to do that using the if statement.

for Loop

The for loop is another example of how Python keeps things simple. The Python for loop can only iterate over a list of numbers. To start from one value, and go to another you use the range function that generates a list of numbers to iterate over.

For instance to print the numbers from 11 to 37 you would use the for loop as follows:

- `for ctr in range(11, 38):`
- `print(ctr)`

The reason we have 38 there is that the range function generates numbers up to but not including the ending number. If you wanted to print every second number in this range you could use range(11, 38, 2).

If you wanted to iterate in reverse, you can specify the range as range(37, 10, -1), although that is ugly and can cause confusions, so it's recommended you instead use the reversed function, as reversed(range(11, 38)).

while Loop
The while loop, like the for loop keeps things simple. The contents of the while loop are repeatedly executed until the condition specified in the loop is no longer true. For example the following will print the numbers 0 to 9.

- `ctr = 0`
- `while ctr < 10:`
- `print(ctr)`
- `ctr = ctr + 1`

Data Structures
There are a couple of important data structures in Python that it is important to know about, and these are lists, tuples, dictionaries, and sets. Let's look at each one of these in turn.

Lists
Python lists are similar to what many languages call arrays. They represent a sequence of values, and can contain a number of types within the same list.

For instance all of the following are valid arrays:

- `arr1 = ["this", "is", "an", "array"]`
- `arr2 = ["so", "is", "this", 1]`
- `arr3 = [1, 2, 3, 4]`

To add a value to a list you can use the append method: 'arr3.append(5)'

To remove the last value and return it you can use the pop method: 'arr3.pop()'

Most interesting though are list comprehensions, these allow you to build lists quickly and cleanly. For instance to build a list containing squares of the first 100 numbers you can do the following:

- `squares = [i**2 for i in range(100)]`

Instead of i**2 here we could be calling a complex function and building an array from the result of that.

Like with most programming languages you can access elements of an the list using square brackets. Like what we discussed with strings, it's possible to access sequences of elements of arrays, to skip elements and to reverse elements using array[start:stop:stride]. It's also possible to assign elements this way. So if you wanted to set the second and third elements of an list to 5, and 7 respectively you can do:

```
array[1:3] = [5, 7]
```

Tuples

Tuples are quite similar to lists, in that they are a sequence of values. However where lists are intended to be modified once created, tuples are sequences that cannot be changed once created. While the use of this might not be readily apparent to you it's important to know that they exist and how to use them because they occur quite commonly in Python.

A tuple can be created the same way as lists but by using round brackets instead of square ones. For instance:

```
position = (5, 7)
```

One immediate use of this is to return multiple values from functions.

Dictionaries

Python dictionaries or dicts allow you to create associative arrays / hashes that can have arbitrary (though constant) values as indices instead of numbers. For instance if we are processing some text and want to count the number of times a particular word occurs we can use dicts as follows:

```
d = {}
d['the'] = 100
d['a'] = 125
d['ophthalmologist'] = 1
```

Note that while we have used strings here, the index of a dict could also be a number or even a tuple!

Another way to define the above would be:

```
d = {
      'the': 100,
      'a': 125,
      'ophthalmologist': 1
}
```

If you just wanted a list of the keys ('the', 'a', 'ophthalmologist') of a dict, you can use the method keys as: 'dictvar.keys()'

Likewise to get the values you can use: 'dictvar.values()'

You can also use 'dictvar.items()' to get a list of tuple containing key-value pairs.

Sets

Sets are once again very similar to lists, but with the restriction that each value may only be added to it once. It can be quite useful for getting the unique values in a list. To convert an existing list to a set, you can use simply pass it to the set constructor method as a parameter. So set([1,2,2,3,4]) will return {1,2,3,4}

Sets can also be incredibly useful in finding intersections, unions and differences between multiple lists.

To create a new empty set use the following:

```
s = set()
```

You can now add elements to this set using s.add(value).

Functions

Functions is Python are similar to most other languages, but support one important feature that makes them quite pleasant to use. That is their ability to specify parameter names. Let's look at a simple function that prints a string a specified number of times:

```
def repeat _ print(string, count=1):
    for _ in range(count):
        print(string)
```

There are a lot of interesting things going on here, first of all this function supports two parameters, 'string' and 'count'. The default value of 'count' is set to 1 so this parameter can be omitted to print a string once.

Second of all we are using the underscore as a variable name in the for loop. This is common practice when we don't care about the value of the variable and will not be accessing it. Here we just need the loop to run a set number of times. You'll also notice that if we just provide a single number inside the range function, it will simply count to that number from zero which works for us here.

We can now easily call this function as 'repeat_print("string to repeat", 5)', but other interesting ways to do the same are as follows:

```
repeat _ print("hi", count=4)
repeat _ print(string=”hi”, count=4)
repeat _ print(count=4, string=”hi”)
```

All three of these will print the string ‘hi’ four times. By providing the name of the parameter we can skip providing values for unnecessary parameter, and we needn’t remember the order of the parameters as long as we remember the names.

Classes

Finally we come to object-oriented development using Python with classes. We’ll jump right into things with classed by defining a somewhat functional class to represent complex numbers.

```
class Complex:
    def _ init _ (self, real, imag):
        self.real = real
        self.imag = imag

    def _ repr _ (self):
        return ‘{} + i{}’.format(self.real, self.imag)


    def add(self, other):
        real _ sum = self.real + other.real
        imag _ sum = self.imag + other.imag
        return Complex(real _ sum, imag _ sum)
```

All our class can do is initialise a complex number, and add two complex numbers, but it’s enough to understand the basics of classes in Python.

The first is the _init_ method. This is the constructor of the class, i.e. the function that is called when a new object of this class is created. The parameters that this method gets are the initialisation parameters for this class. Methods in Python are provided an additional parameter (called self by convention) that gets the value of the object when called. This is similar to the ‘this’ variable present in languages like C / C++ etc but feels a lot less magical since it is explicitly visible.

The constructor method simply stores the provided real and imag parameters within the object by adding them as properties of ‘self’. They can then be accessed by other methods in this class.

The second interesting method is _repr_. This method is designed to returned a string representation of a complex object. If we did not write this function then printing an object of the Complex class wouldn't show very useful information when printed to screen.

Finally the add method takes as parameter another complex number (we just assume here, in real code it should check for this) adds it to itself and returns a new complex number that is the sum of these numbers.

Note that while variable and method names still use Snake Case in Python, class names are in CamelCase by convention.

Now let's look at class inheritance by creating a subclass of this complex class but one that adds a feature.

```
class Complex2(Complex):
    def _ _init_ _(self, real=0, imag=0):
        super()._ _init_ _(real, imag)
        def conjugate(self):
        return Complex2(self.real, -self.imag)
```

This class derives all the functionality from the above Complex class, but adds an additional method that returns the conjugate of this complex number (same real part, negated imaginary part). Additionally it also improves the constructor by setting default values of 0 for the real and imaginary parts if no value is provided.

The new _init_ method simply passes along the values it gets to the _init_ method of its superclass (Complex). The original _init_ method now handles storing the values as properties.

Virtual Environments and Python Packages

Often when working on a Python application you will need to use libraries created by other people. Python has great tools for finding and installing such packages, and then importing them into your own code to use the ready-made functionality. Django, which we will use later to build our CMS is one such package.

There can often be conflicts between different pieces of software that need different versions of the same libraries to work. To avoid this the virtualenv package was created, and it is now available by default with Python 3.

This package creates a virtual Python install inside a folder that once activated appears to application as the default Python install. Any packages you install while this virtual Python install is activated are installed within this folder rather than globally for the entire system.

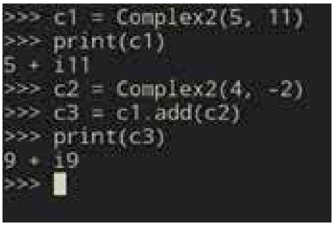
When we develop our CMS using Django it would be wise to create a separate virtual environment for any packages that we install; although this is not necessary.

To make a virtual environment with Python 3 you can use the following command:

- `python3 -m venv project`

Note here that for Windows you might need to use 'python' or 'C:\Python34\python.exe' instead of python3.

This will create a new directory called 'project' (or whatever name you used) with the virtual Python install. To use this Python install you need to run 'source bin/activate' within this folder, or 'Scripts\activate.bat' on Windows. If you prefer PowerShell on Windows run 'Scripts\Activate.ps1' although you might need to enable script execution for this to work.



```
>>> c1 = Complex2(5, 11)
>>> print(c1)
5 + i11
>>> c2 = Complex2(4, -2)
>>> c3 = c1.add(c2)
>>> print(c3)
9 + i9
>>> 
```

python-classes

To install Python packages, you can use the 'pip' command. On some Linux distributions you might need to install this. With pip you can install packages as follows:

- `pip install package _ name1 package _ name2`

You can also create a text file that has all the packages you need listed in it, and then install them all at once as follows:

pip install -r requirements.txt

This text file is by convention called requirements.txt

Once you have all your packages installed, you can use them in your own code by using the import command. For instance to use the popular `mutagen` audio metadata manupulation library for Python you first need to install it with `pip install mutagen` and then you can use it as follows:

- `from mutagen.easymp3 import EasyMP3`
- `mp3 = EasyMP3("audio _ file.mp3")`
- `print(mp3.tags['album'])`

This will print the album for the `audio_file.mp3` mp3 file.

Conclusion

There is a whole lot more to Python than we could get into here, but hopefully you now know enough to get though this book with ease.

STARTING WITH DJANGO

Django has been around since 2005 when it was created by developers for a newspaper's website. Now it is an open source project with its own foundation running thousands of websites

Django is a Python-based web framework that touts itself as "The web framework for perfectionists with deadlines." This should tell you at least a little bit about what the framework focuses on.

It include a large amount of common functionality that is required for any website to function. Defining how URLs work, how data is structured in the database, the layout and design of the website and a lot more is possible using just Django itself. In fact this is often a criticism of Django, that it includes too much functionality within a single monolithic package rather than providing a looser set of interacting modules. Unless you are only using a very small part, perhaps only a single feature or two of Django though, this needn't be much of a concern to you.

It would be entirely possible in fact to use pure Python for building a CMS application, using single independent modules for the functionality that Django provides, in some cases this might even be the better solution. However when building a CMS, you are likely to use a lot of features that

Django provides. Such is the case with our application which takes advantage of a lot that Django offers. Not all of it of course, but whatever we don't use can be disabled and mostly doesn't affect the performance.

Django also happens to include common sense security mechanisms that protect you from CSRF (Cross Site Request Forgery), XSS (Cross Site Scripting), SQL Injection, Clickjacking, etc. It abstracts the database you use so you can use a single-file SQLite database during testing and development and move to Postgres, or MariaDB, in production without needing any code changes in most cases.

All of this makes Django well suitable for building our CMS from scratch with a good feature set, in minimal time, but without compromising on security.

Installing Django

There are many ways to get Django but the most straightforward and recommended way is to install it using `pip` the official Python package manager.

With pip installing Django is as simple as running:

```
pip install django
```

Running the above in a command shell will install Django for the current default Python version. If you are using virtual environments as recommended in the previous chapter, it will install within the virtual environment and leave your main system clean.

There are actually a number of different versions of Django that you can currently get and install. Every so often a Django release is marked as LTS or Long Term Support, and this release is supported and patched for a longer period (3 years) than usual. Currently both the previous LTS version (1.4) and current LTS version (1.8) are supported, and the previous version of Django (1.7) is supported as well. For our purposes there is no reason to use any version other than 1.8 and that is the version that will be installed when you run the command mentioned before. If you are reading this booklet a long time after it came out, chances are a new version of Django is out. The CMS we developed here should still work, however if you face issues you can force pip to install a specific version of Django by specifying its version number as follows:

```
pip install django==1.8.3
```

Here 1.8.3 is the current version of Django. A better way to do this though is to use the following command:

```
pip install "django<1.9"
```

The quotes are important here, you need to type them! This will install the latest version of Django that came before 1.9, which means you will get the latest patched version of Django 1.8

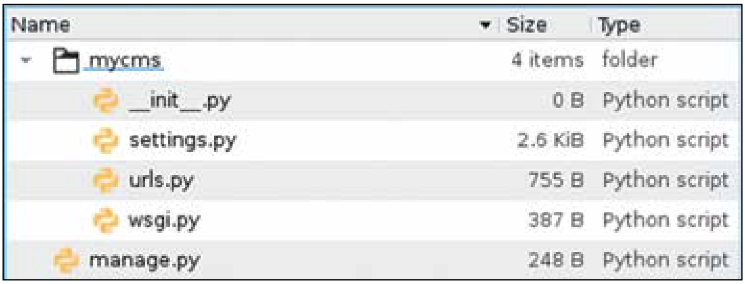
Starting a New Django Project

Django projects need to follow a specific project structure, and Django includes the 'django-admin' command that amongst providing many other features also includes a subcommand 'startproject' to create this structure for you.

To create a new project called 'mycms' in the current directory, run:

django-admin startproject mycms

This will create the following project structure:



Name	Size	Type
mycms	4 items	folder
__init__.py	0 B	Python script
settings.py	2.6 KiB	Python script
urls.py	755 B	Python script
wsgi.py	387 B	Python script
manage.py	248 B	Python script

django-newproject

Let's look at this structure in detail.

The newly created 'mycms' directory contains two entires, a manage.py file and yet another directory called mycms. The 'manage.py' command allows us to perform administrative tasks on our project. The 'mycms' directory contains configuration code for our project.

The 'settings.py' file, as its name suggests, contains the configuration settings for our project. In this file you will find configuration parameters such as your project path, template and static file paths and urls, your database settings and included applications, time zones etc. For our purposes most of the default settings will work just fine, including the default database settings.

By default Django applications are configured to use a file-based SQLite3 database for storing data. You can see this setting defined as 'DATABASES = ...' in the settings file. Note the plural, Django can use multiple databases of different types in the same app!

By default all the data your site will create will be placed in a file called 'db.sqlite3'. You can open this file in any tool that can explore SQLite3 databases and see how it's structured. You can also change the name or location of this file by changing the 'NAME' parameter in the 'DATABASES' setting variable.

There are some parameters that you will need to configure to be able to upload files to our CMS and to display them on our web pages, we need to configure the directory where these files should be uploaded, and the URL on the site where they should be served from. These settings are as follows (feel free to change them to something that works for you):

- `MEDIA_ROOT = 'D://projects//mycms//media//'`
- `MEDIA_URL = '/media/'`

The 'urls.py' file configures how different URLs within our site are handled. The 'wsgi.py' file sets up a WSGI application for our project allowing us to serve it using a WSGI server, you don't need to touch this file at all. The '_init_.py' file, if present in a folder marks that folder as a Python package that can be imported rather than a just a directory containing Python files.

This sets up our Django project, which represents a website. This project in turn is composed of a number of applications. For instance while we are building a CMS, we could also include a forum app and a wiki app in our project if we need those in our project. There are also ready-made applications available that you can install using pip and then use in your project directly.

Most of the code we will write for our CMS will be part of an application we create for our project. Our next step is to create and set up our cms application within this project.

Setting up a Django Application

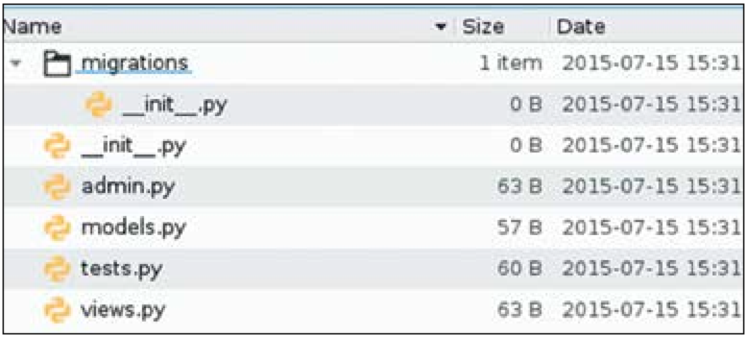
A Django app encapsulates a single module of functionality, such as a polling application, a forum, a blog, a gallery, etc. In our case it is a more general CMS.

You can create a new application for our project using the following command:

```
django-admin startapp cms
```

You need to run this command inside your project directory, i.e. the directory containing 'manage.py'. In fact you can use 'python3 manage.py' ('python manage.py' on Windows) instead of 'django-admin' here. This will create an application called 'cms' for our project, and it will appear as a directory called 'cms' within our project folder along with the 'mycms' directory and the 'manage.py' file.

Let's examine the structure of this directory:



Name	Size	Date
migrations	1 item	2015-07-15 15:31
__init__.py	0 B	2015-07-15 15:31
__init__.py	0 B	2015-07-15 15:31
admin.py	63 B	2015-07-15 15:31
models.py	57 B	2015-07-15 15:31
tests.py	60 B	2015-07-15 15:31
views.py	63 B	2015-07-15 15:31

django-newapp

admin.py: The code that sets up the administrative interface for our website

models.py: Database models for our website

tests.py: Here you can write unit tests for your application

views.py: The views contain code that renders the actual content that will appear in a user's browser

init.py: As mentioned before, this makes the 'cms' directory a Python package, ready for importing

migrations: As you make changes to the structure of your database, Django keeps track of the changes made in this folder so you can go from an older structure to a newer structure without losing data.

Now that we have our 'cms' application set up, we need to include this application in our project by adding it to the appropriate place in the 'settings.py' file. Open the 'settings.py' file in the 'mycms' directory and look for the 'INSTALLED_APPS' setting that contains a tuple of installed applications. By default it will include some of the applications that are distributed with Django. We need to add our newly created 'cms' application to this list.

Whenever you modify your application in a way that affects the database, you need to do two things:

First, you need to run the 'python3 manage.py makemigrations' command to tell Django to look at what has changed in your application and create a migration file that defines how to go from the previous structure of your database to the new structure.

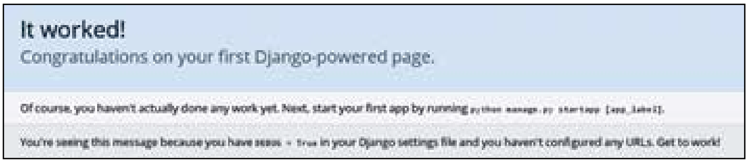
Second, you need to actually apply these migrations to your database so it can work with your new code. This can be done with 'python3 manage.py migrate'.

Since we have just created a new project, and a new application, the state of your database differs from the state of your code. So you will need to run both these commands (although since out app is currently empty, 'makemigrations' will do nothing).

At this point you are ready to start running your application! Just run the following command to start your application:

- `python3 manage.py runserver`

This will run the inbuilt development server included with Django. You can now open your site in a browser by entering *http://localhost:8000/* in the address bar. You should see a simple page containing a message with the heading 'It worked!' One great feature of this command is that it checks if you have monitors your code for changes and automatically restarts itself if you, unless your code has errors that cause it to break.



It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

The default greeting page for Django

The Main Components of DJango

Django is a complex piece of work that works hard to keep your own code simple and easy to follow. Let's look at some of the major components that make up Django and how they will fit in our application.

The Object Relational Mapper

Django includes an Object-Relational Mapper, or ORM that simplifies how you deal with your database. An ORM maps database entries to corresponding objects in a programming language, thus allowing you to use a much more natural means of querying, updating and otherwise interacting with the database than something like SQL.

The Django ORM allows you to think of your data as objects, as instances of a Python class that you can access, modify, and delete rather than as rows in a database table. In the background, Django translates all this to the equivalent SQL code and runs it on your database server.

This also makes your interaction with the database agnostic to the actual database technology. Django supports MySQL / MariaDB, Postgres, SQLite or Oracle Database Server out of the box and can support others via third-party packages. It doesn't matter which one you use for the most part, the ORM will work in a similar manner.

Django's ORM has a powerful query syntax that allows you to search, sort and filter the contents of the database to get to exactly the entry or entries you want. It is also capable of efficiently combining multiple related tables, so it appears like you are using a single object while the data is coming from multiple tables. It also simplifies creating new database entries and updating exiting ones, whether one-at-a-time or in bulk.

Recent versions of Django can also track changes you make to your database structure during development, and will automatically generate code that will modify a database, taking it from the old structure to a new one without loss of data.

Django can also interface with multiple different databases at the same time in the same application.

Views

Django views are the bits of code that use the Django ORM among other things to create the response that the user will see when they browse a page.

Django supports two kinds of views. The old function-based views are simply a single function that can receives a request, crunches some data, and returns a response. The newer class-based views are more structured and offer common base-classes that encapsulate common functionality, such as displaying a single object, or listing objects of a particular type.

That isn't all when it comes to views, since they also need to handle data creation and editing. For instance you might have a registration form to let users sign up for a new account. Or you might have a contact page where users can leave you a message. These situations need you to access the data the user entered, verify and validate it and then add it to the database if needed.

Django provides tonnes of tools an utilities for all possible cases such as blocking users from accessing certain pages if they haven't logged in, or from posting data unless they have the correct permissions.

Views need not even be only about generating HTML content. You may also want to provide an API for your website that allows developers to access your data as JSON or XML. Or you might want users to be able to export their personal data as a CSV file. Another possibility could be to allow users to save articles in PDF format.

Django helps you develop views for all these use cases and more.

Templates

Django includes its own templating language that can be used to generate HTML code for your web pages.

With Django's template language you can write your frontend HTML code as normal, and simply use special template block that designate where data need to be filled in. Then in your view functions you can combine this template with the required data and return that as a response to display in the browser.

The Django template language is quite simple, but still quite powerful and extensible. One of the most useful features of this language is the ability to extend one template from another, and to import common template code.

For instance you can create a base template for every page in your website to follow. This template can contain the basic JavaScript and CSS code for your website. You can then extend this template to create a base template for all your article pages, another base template for all your gallery pages, etc. Thess can further be extended based on your need.

We also mentioned how it is possible to extend Django the template language. In fact Django itself ships with some convenient extensions that can be enabled if needed. For instance Django ships with an app called humanize that simplifies conversions such as formatting numbers with commas as separators, converting numbers like '4500000' to '4.5 million', converting absolute dates to relative dates etc.

The template language used in Django is designed to have minimal logic, and no complex code. Any processing you need to do should happen in the view and the template should merely get the final data it needs to display.

As of Django 1.8 it is simpler to plug in other third-party template languages in Django; however since the view has never been restricted from using other third-party code it was always possible to use any Python-based template language to generate your responses.

Forms

Often times you need to get just a few details from a user, such as their email, name and feedback for a feedback page, or email, name and message for a contact page. Or you might need their personal details for registering an account.

For all these situations and more Django has as the forms framework. Django's form framework allows you to not only create such forms for displaying on your website, but also supports creating automatically creating forms for existing models in your database.

Django forms framework handles cleaning and validating the data entered by the user, and provides the facility to customise this validation to include your own logic.

Authentication

Another useful and powerful framework that Django includes out of the box is the Authentication framework. This framework includes models for Users and Groups, and includes a permissions system that allows adminstrators to control what users, and what groups are allows to do what kinds of tasks.

These systems are flexible enough to apply to any kind of content that you create with Django. For instance you could create a CMS where regular users can create and edit articles, but not delete them, where they can create tags, but not edit or delete them, and where they can apply categories to articles but not create new categories.

You can also create your own kinds of permissions. For instance on a website that charges for viewing some kinds of articles you might want to add a permission to view articles and only give that permission to users that belong to the group of paying members.

Of course you will still need to check for this permission before displaying those articles; that is not automatically handled as Django has no way of knowing what your permission implies just by what you name it.

We won't be doing much with the permissions framework in our CMS, but it is an important component in any good CMS, and we are getting it for free by using Django.

Emails

Whether you want to notify users of new articles, or want to sent a newsletter or to share articles with others via email, there is a good use case for having the ability to send emails from your CMS.

By combining the ability to generate PDF files with the ability to send emails, you can even add the ability to send articles to a user's Kindle device by sending an article's PDF content to their Kindle email address.

There is a lot you can still do with email and Django has great support for email. All you need to do is to configure an SMTP server, and then you can send mails with a simple function call.

You can send both HTML and plain-text mail, and can even include attachments. The template framework can also be useful in generating the contents of your email.

Some Other Important Modules

Django still has a lot more to offer, and we won't go into all of that here. However there are a few other bits of Django that we would like to go over:

Serialization Framework

This nifty little framework makes it simple to convert data from your database into the equivalent XML or JSON representation. It's also possible to create your own serializer to convert to custom formats.

This framework can also go the other way, and convert an XML or JSON representation of data to a model instance that can be added to the db or used to update existing data.

Sessions Framework

Django inludes support for tracking user sessions, which enables users to log in and stay authenticated. It also simplifies storing data associated with a user using cookies, so you can track products that a user has added to their cart, or products the user has selected to compare etc.

Static Files Framework

There are a lot bits of your website that are not being generated by Django. For instance CSS and JavaScript files, images and videos that are part of your design, fonts and vector artwork etc. These are called static files since they are not dynamically generated and Django can handle serving these files to the end user as well.

However this application is designed to be used only while testing the website, not when used in poduction. In production you need to use a proper web server like Apache *httpd* or Nginx to serve these files.

Pagination Framework

Unless you like the idea of listing all of the thousands of articles in your website in a single page, chances are you will want to use this framework to paginate your data.

Syndication Framework
If you'd like to provide RSS or Atom feeds of the contents of your website, the syndication framework in Django will be very useful.

The Anatomy of a Django Request
We just launched a server for our application and got something to display in a browser, but how did it all happen? Let's follow the working of a Django application from request to response.

Let's see what would happen if our application were to get a request to *http://localhost:8000/testing*

The first thing Django does is to look in the 'urls.py' file in our project folder to see if we have any pattern matching '/testing'

The patterns in 'urls.py' are each associated with a view that handles what should happen if the URL matches that pattern.

When a pattern is matched, Django runs the code corresponding to that view, and passes it all the information that the user provided in the request (for instance the user may have submitted a form, or might have cookies stored)

This code can then do pretty much anything! It can use any Python library or feature, it can launch other applications on your system to get a response. Realistically though it will look up some information in a database, convert into into some pretty HTML code and return that to the user.

The user can then see this code in their browser!

Simple enough right?
There are however more steps along the way that we have left out because we will not be using those features. However Django also includes code called middleware that provide security features or other important features along the way.

DESIGNING OUR CMS

It's always good to start with a road-map, so we know how far we've come and know when we are done

Now that we have some idea how Django works, before diving into the actual coding, let's discuss what all we want our CMS to do, and how that maps to different features of Django.

Let's start with the most important thing any CMS contains, and that is content! In our CMS we will support a couple of different kinds of content.

Content Types in our CMS

Articles

In our CMS an article will have the following information associated with it:

- A title
- An Author
- The actual content
- A featured image
- Publish status
- A category
- Tags

These are the fields that a user will enter for each article. However there are also some fields that our CMS will track that are going to be automatically generated, and in some cases not even visible to the user adding the article. They are:

- A slug: This is a URL-friendly version of the title.
- Creation date: The date the article was created
- Date of last edit: The date and time the article was last edited

Since Django is a bare-bone framework for building all kinds of websites, it doesn't include an in-built way to work with articles written in HTML. So instead we will use the popular Markdown syntax for writing articles in plain text, which will then be rendered in HTML before being displayed.

Categories

A category is a much simpler concept than an article, although we will spice it up a bit. Our categories will have the following:

- A name
- A slug
- A short description
- An image

Each category can then have its own page showcasing the image, and description for that category followed by the articles in that category.

Tags

Tags are possibly the simplest piece of content in our CMS. A tag is composed of simply a name and a slug. The slug is required because tags might contain special characters that can't be a part of URLs.

Authors

While there is a lot we can do with authors such as making them editors, administrators, in our case we will just use the inbuilt authentication framework that Django includes. Django's auth system is quite flexible and powerful and can already do what is needed by us.

With Django's authentication system you can add users, and groups and assign them permissions. For instance we could create a group called 'Author' that only allows one to add and edit articles but not delete them. A 'Manager' group could then be created that could add, edit and remove articles. You can also create custom permissions for more fine grained control.

It is possible to add the ability to restrict editing of an article only to its creator, or to 'Managers' / 'Editors' however we wont go into that in this text.

Image
Our CMS will need to have the ability to upload images so they can be included in articles. As such we will give users the ability to manage image files. Each image file will also need to store associated information such as:
- Name
- Slug
- Alternate text
- Image file

Gallery
A gallery is just a collection of images. As such it will need to store the following information:
- Name
- Slug
- Description
- Included images
- Layout

The final bit, the layout is something we are adding just for fun. We will add the ability to select a layout for your gallery page. For instance we can have a gallery that has thumbnails linked to the image page, another layout that has all images displayed on the same page with the description next to them, and a third layout that has each image with a description below it.

An Admin Panel
Another major feature we will need on our website is an administration panel that allows us to manage the different types of content.

The good news here is that Django does most of the work for us here by proving an administration application that can automatically build an interface to add, remove and edit all kinds of content based on our description for it.

This panel is very customisable allowing us to specify how to list different types of content, how to filter and search within content and how to lay out the different pages that manage this content.

Different Ways to Browse Content

We've defined a lot of different kinds of content for our website, but it is meaningless if we cannot browse this content in some way. All the categorisation and tagging is useless if they don't make it easier to discover content on our website.

The first and foremost way to discover content on this CMS will be to just visit the front page. This will list the top two categories on our website, the 50-word summaries of the latest two articles, and links to the next eight, and links to browse the website in different ways.

We will include a page for each category so users can browse a list of articles for a single category. Additionally we will have a page that lists all the categories as well.

Similarly we will include a way to browse articles based on a tag in the article. Like categories we will include a page that lists all the top tags in the form of a tag cloud.

We will also provide a page that lists all the authors on our website, and have a page for each author that displays the articles by that author.

There will also be a page for each image that displays an image, its tags, and its description. There will not be a page to list all the images, but there will be one to list all the galleries, and a page for each gallery that uses the chosen layout.

DJANGO MODELS

Django's ORM is one of its most powerful features and it can benefit your website a lot if you know how to use it effectively

We talked a bit about Django's Object-Relational Mapper way back in the 'Starting with Django' chapter — seems ages ago now doesn't it! Now we actually use it in our project by designing how the CMS will structure its data in the database. In the previous chapter we talked about this structure, and in this chapter we turn that to code.

Before we do though we need to understand how the Django ORM works and how we can use it in our CMS.

Anatomy of a Django Model

A Django model is the Python-equivalent of a table in a database. A database table generally holds information about a single type of data, and Django allows us to configure the structure of this table using Python code.

To create a model in Django we simply need to create a Python class that derives from (i.e. extends / subclasses) Django's Model class and place it in the 'models.py' file in the app directory (in our case 'cms').

This class will have a number of attributes, each of which represents a column in the database. We can specify what kind of data is to be held in that attribute / field and that will decide the kind of database column that will be created to store that data.

We can also specify some ‘Meta’ options for the class that will change the way the class behaves in some cases. For instance we can configure extra permissions for this model in the meta options. We can also specify a name for the database table for this model if you don’t like the one that is automatically generated.

Let’s create a model for our Article class to see this in action. The following code for ‘models.py’ contains a model class that showcases fields, metadata and methods:

```
class Article(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    author = models.ForeignKey(User)
    content = models.TextField()
    creation_date = models.DateTimeField(auto_now_
add=True)
    edit_date = models.DateTimeField(auto_now=True)
    featured_image = models.ImageField(null=True,
blank=True)
    publish_status = models.BooleanField( default=False)

    def __str__(self):
        return self.title

    class Meta:
        verbose_name_plural = ‘Articles’
        ordering = [‘creation_date’]
```

The above code creates an Article model with:

- A character field for the title that with a maximum length of 255 characters
- A slug field that will simply have a cleaner version of the title
- An author field that uses Django’s ForeignKey field to link to a ‘User’ which is another model and another database table
- A content field that is a TextField for large volumes of text.
- A creation date that is automatically set to the time and date when the article is added
- An edit date that is automatically set to the time and date whenever the article is edited
- A featured image field, that has ‘null’ and ‘blank’ set to true, which

marks it as an optional field that can be left blank and can be stored as null in the database

- A publish status field that can be toggled between published / unpublished, it default to False, or not published

Note that we have used an 'ImageField' in our model, and that particular feature of Django requires that we install a Python library called Pillow. So go ahead and run 'pip install Pillow'. On Linux systems this might require you to have a development environment set up. If pip fails to install Pillow, check your distro's repositories for a version of Pillow for Python 3 and install from there.

You will also notice a '_str_' method that simply returns the title of the article. If we omit this then Django does not know how to give this model a name, and thus by default uses names that are unhelpful. Here we are telling it that the title of the article is what this piece of content should be called.

The Meta class inside the model class sets up two properties. The first tells Django that the plural of Article is Articles. In this case it is entirely unnecessary since Django is quite capable of doing this itself, it can even handle pluralising some common words where the plural isn't simply the singular with an 's' in the end. You need only do this if you notice that Django has got the plural wrong. The second property in the Meta class tells Django that this model is to be ordered based on the creation date of the article.

One of the hallmarks of relational databases, are well, relations. We have seen a bit of that above with the ForeignKey field that links the Article to a User, but relations between different tables are something that we should probably discuss in a little more detail.

Relations Between Models

The way relational databases, like the ones that use SQL store complex hierarchical data is through the use of relations. One piece of data in one table can have further pieces of data in other tables associated with it. The way these are tracked is by giving each piece of data a unique ID that can be referenced by other pieces of data. Let's look at this through a few examples.

Imagine we want to store information of different companies and their employees. Now each company can have multiple employees, but each employee can only be part of one company. This kind of relation going from a company to its employees is called a One to Many relationship, since we have one company associated with many employees. Going from employees to company it would be called a Many to One relationship, as many employees can have one company.

The way this would work with a database is that each company would be given a unique ID, called its primary key. Often this is just a serial number that increases with the addition of each company. Employees likewise are given a unique primary key ID. In the table were we store data about individual employees we also have a column that references this company ID.

So if you encounter a company with a unique ID for 17, and want a list of all its employees, you would simply ask the database to fetch a list of all employees that have a company ID equal to 17. Likewise if you look up an employee and want to know the name of the company they work for, just ask the database to pull details about a company with a unique ID that is equal to the ID stored in this employee's company column.

The way we can implement this in Django is through the ForeignKey field. So having a ForeignKey to a company in an employee model is us telling Django that there is a Many-to-One relationship between employees and companies. This is what we have also done with articles and users in our article model.

There are other types of relationships that data can have though. For instance we can have a Many to Many relationship as well. Consider the relationship between customers and companies. A company has multiple customers — unless there is a monopsony — and a customer can be a patron of multiple companies — unless there is a monopoly.

The way this would be implemented in a relational database, is through the use of a third table that defines the relationship between companies and customers. This table would only need to have two columns, a company ID and a customer ID. Then we can create a link between a company and a customer by having a row in this table that has their IDs in it.

Now if we want to look up all the customers of a company with an ID of 29, we tell the database, look up the IDs of all the customers that have an entry in the customer-company link table where their company ID is 29; and for each of these IDs look up the customer's details in the customer table.

We could as easily go the other way. For a customer with ID 1973, look up all the companies IDs that are linked to the customer ID 1973 in the customer-company link table, and for each of these companies fetch the details from the company table.

In Django this is modelled using a ManyToManyField. Tags for instance can be associated with multiple articles, and an article can have multiple tags. So we need to link an Article with its tags using a ManyToManyField in the

Article model. Django will automatically create the table to link between tags and articles and use it to fetch tags for an article and the other way around.

You'll be interested to know that it is also possible to have One to Many / Many to One and Many to Many relationships within the same model / data! For instance employees can have a boss, who is an employee as well. There is a many to one relationship here since many employees can share the same boss. Likewise, if we track friendships between employees, then one employee can have many other employees as friends, which is a many to many relationship from employees to employees.

There is one other kind of relationship possible between data, and that is a One to One relationship. An example of this could be an employee profile. A single employee will only have a single profile and that profile will belong exclusively to that employee. Of course it would be possible to just have the profile data included within the employee table itself, however sometimes this can be useful for efficiency, lesser used and accessed data can be kept off in a separate table.

Testing out our new Model

Now that we have a new Model class in our application we can see test how it works in the Django shell. The Django shell is a convenient tool provided by 'manage.py' that simply loads a Python interpreter that is properly wired to work with your Django project. You can start this shell by executing 'python3 manage.py shell'

Remember though, what we mentioned in our 'Starting with Django' chapter. After any change to the models, you need to run 'python3 manage.py makemigrations' followed by 'python3 manage.py migrate'

Now that you are in the Django shell you need to import your Article model to use. You can do this by typing:

```
from cms.models import Article
```

Now you can use the `Article` class to access the list of articles and to create new Articles. Since the database is currently empty, let's create a new Article. In the shell run the following commands, which will create a new blank article, fill in some of the details and then save it to the database:

```
>>> article1 = Article()
>>> article1.title = "New Article"
>>> article1.content = "Test content"
>>> article1.slug = "new-article"
>>> article1.save()
```

Were you greeted by a long list of errors culminating in something along the lines of 'django.db.utils.IntegrityError: NOT NULL constraint failed: cms_article.author_id' We caused an error in Django's integrity, not nice.

What this error boils down to in the end is that we tried to create an article without specifying an author, and the author is not an optional field, unlike the article featured image, which we also did not supply.

The problem now is that we do not have any users right now! Our database is fresh and empty. No problems, we can just create a new user just as we would create a new article. Run the following commands in the shell:

- `>>> from django.contrib.auth.models import User`
- `>>> author1 = User()`
- `>>> author1.first _ name = 'Aruna'`
- `>>> author1.last _ name = 'Tank'`
- `>>> author1.email = 'aruna.tank@mailprovider.com'`
- `>>> author1.set _ password('apassword')`
- `>>> author1.save()`

This should work out a lot better than out experiment with creating a new article did. Notice that we used a method to set the password rather than just saying 'author1.password = "xyz"', this is because passwords should never be stored directly as raw text in the database. The 'set_password' securely saves the password in the database. If you want to know the exact value it stores in the database, just type `author1.password` in the shell to see this value.

Now that we have an author, let's trying saving that article again. If you didn't close the shell in between you can still access the partially complete 'article1' else you will have to type all of that all over again. Here is what you need to do:

- `>>> article1.author = author1`
- `>>> article1.save()`

This should now work without a hitch. We now have one author and one Article in the database! Go ahead and create more Articles and Authors. It'll be nice to have a few in place so the next bit is actually meaningful.

Querying our Database

In this section we learn how to query our database so we can get exactly the data we want. Of course this will only be worthwhile if there already is some data in the database that we can query! And the more data the better.

For this reason we have created a JSON file that comes filled with a collection of Users and Articles. You can import this into the database to populate it with content that can be queried. You can download this file from the following URL:

https://raw.githubusercontent.com/9dot9Media/fasttrack_to_cms_code_snippets/master/05_django_models/test_data_1.json

You can then import it into your database using the following command:

- `python3 manage.py loaddata test _ data _ 1.json`

Django supports some very advanced data queries and manipulations that we wont go into in this text. We will however look at all the basic types of queries that are essential for nearly any application.

For our article list page, we will probably need to display a list of all articles. The way to ask the database for all Articles is:

- `Article.objects.all()`

Similarly if you want all Users / Authors:

- `User.objects.all()`

Actually, if we think about it, we don't want to display all articles. After all we have an keep the publish status for each article for a reason, we don't want articles not marked for publishing showing up. So we need some way to filter for only those Articles that have been published. This is easily accomplished with the following query:

- `Article.objects.filter(publish _ status=True)`

Likewise we can provide 'publish_status=False' to get all unpublished articles. Another way to get a list of all articles that have their publish status set to true is to exclude all articles that have their publish status set to False, since it is a binary option. Here is how to do that:

- `Article.objects.exclude(publish _ status=False)`

Let's find out how many of our articles start with 'The' shall we. This query can also be accomplished using the filter method:

- `Article.objects.filter(title _ _ startswith='The')`

Simple right? Now let's make this a little more complex. How about we get a list of all the articles written by authors whose first name starts with a W? We don't know why you'd want to know this, and perhaps it isn't out business to know, but here's how you can get to that sweet data:

- `Article.objects.filter(author _ _ first _ name _ _ startswith='v')`

Note here that the first name of the author isn't even stored on the same model, it is part of the User model which is stored on a different database table! Django just automatically manages this for you. Since we are looking up a field of model linked via the author field in the Article we need to use double underscores between author (a field on the Article model), and first_name (a field in the user model). In object oriented parlance this is equivalent to article.author.first_name.startswith('v').

There are a lot many more ways we can filter and look for the content we need. Like the 'startswith' trick we used above, you can also use creation_date__gt to check for publish dates greater (hence gt) than the supplied date, or __lt for less than. For example filtering with : 'creation_date__lt='2014-02-14'' will return all articles that were created before February 14th 2014.

You can also supply multiple filters at once, or chain filters. So the following both return the same result, articles with a title that starts with 'A' that are not published:

- `Article.objects.filter(title__startswith='A', publish_status=False)`
- `Article.objects.filter(title__startswith='A').filter(publish_status=False)`

The above are both equally efficient. You can even chain an exclude after a filter.

You can also sort / order the results based on one of the fields using the 'order_by' method. For example:

- `Article.objects.order_by('title')`

The above will order the articles by their title. Of course you can follow this up with additional filter calls or use the 'order_by' method after filter calls.

You will also be pleased to know that the result of all of these operations behaves like a Python list (although it isn't one). So you can use the familiar list operations such as accessing a particular index with '[index]' or a range as '[start:end]' or even a strided range with '[start:end:stride]'.

Modifying Data

Now that we have a bunch of data in our database, we should can go about modifying it and saving it back to the db. This is as simple as querying the db for a single object, editing its properties and using the `save` method just as we did while creating a new entry!

If you know the id of any of your articles, you can fetch it using the 'get' method as follows:

- `article2 = Article.objects.get(pk=11)`

The above will fetch the article with an pk (primary key aka id) of 11. Now you can modify the title, contents or any other field just like before, and then use the 'save' method. Let's make things a little more interesting then, let's modify multiple articles at the same time.

Let's saw we have a lot of faith in author's whole name starts with a 'K' — why wouldn't you — and we'd like to publish all articles written by such authors. We know how to write this query, we did that in the previous section:

```
Article.objects.filter(author__first_name__
startswith='K')
```

A brute force way would be to use use a loop to go though this list of articles, individually set the 'publish_status' to True and then save the article. However there is a better way, and it is as simple as the following:

```
Article.objects.filter(author__first_name__
startswith='K').update(publish_status=True)
```

If you execute this query in the shell, you will see a number sow up. This is the number of rows that were affected by this update.

Creating the Models for our CMS

Now that we know how models can be created and used, let's go about creating all the ones we need. At this point the only thing we need to keep in mind is the fact that some of our models need other models in order to work.

For instance, articles need to have tags and categories. In the model for article that we created in the previous section, we left these out. In this section we will update the model we made before to include support for tags and categories. Before that though we need to create models for tags and categories.

So let's start with the simplest, the tag. Here is the model for tags, you should find it entirely unsurprising:

```
class Tag(models.Model):
    name = models.CharField(max_length=32)
    slug = models.SlugField()

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

The categories model is likewise quite simple:

```
class Category(models.Model):
    name = models.CharField(max_length=32)
    slug = models.SlugField()
    description = models.TextField()
    image = models.ImageField()
    def _ _ str _ _ (self):
        return self.name
```

```
Now our Image model:
```

```
class Image(models.Model):
    name = models.CharField(max _ length=32)
    slug = models.SlugField()
    alt _ text = models.CharField(max _ length=255)
    image = models.ImageField()


    def _ _ str _ _ (self):
        return self.name
```

You might be wondering, why do we have a separate model for image that itself has a field that actually contains the image. Why not just directly use the image field wherever needed?

There are two reasons for us to do this, firstly this allows us to store additional data about the image such as its name and some alternate text; secondly we have to have a fixed number of fields in a model, so we cannot have a gallery model that allows an unlimited number of Images unless images are stored as a model rather than as a field.

Now our model for galleries:

```
class Gallery(models.Model):
    name = models.CharField(max _ length=32)
    slug = models.SlugField()
    images = models.ManyToManyField(Image)
    description = models.TextField()
    layout = models.CharField(
        max _ length=5,
        choices=(
            (‘LINK’, ‘Linked’),
            (‘HORI’, ‘Horizontal’),
            (‘VERT’, ‘Vertical’)))

    def _ _ str _ _ (self):
        return self.name
```

Here you should see some things worth talking about. Firstly, the layout field that has a choices parameter which is a tuple of tuples. This is Django's way of defining a choice field, a field that offers users a list of choices rather than a text box to fill details in. This can be useful to give users a choice of countries / states / cities, or in our case a choice of layouts.

The choices parameter has a list of choices to present to the user, where each choice in turn has two entires, the first is the value to store in the database, and the second the corresponding value to show to the user. So in the above case if a user selects a 'Vertical' from the list of layouts, the actual value being stored in the database will be 'VERT'. It is also possible to use an integer field for this, in which case each choice would correspond to a number code.

Another new kind of field we see here is the 'ManyToManyField' used for tags and images.

Finally let's go back to the Article model, now with all fields included:

```
import markdown

class Article(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    author = models.ForeignKey(User)
    content = models.TextField()
    creation_date = models.DateTimeField(
        auto_now_add=True)
    edit_date = models.DateTimeField(auto_now=True)
    featured_image = models.ImageField(null=True,
                                       blank=True)
    publish_status = models.BooleanField(
        default=False)
    category = models.ForeignKey(Category)
    tags = models.ManyToManyField(Tag)

    @property
    def content_html(self):
        return markdown.markdown(self.content)
    def __str__(self):
        return self.title

    class Meta:
        verbose_name_plural = 'Articles'
        ordering = ['creation_date']
```

Remember that we want the content of our articles to be typed in Markdown and then converted into HTML before being rendered on a page. For this we are importing the Markdown library and using it to convert the article contents to HTML.

The Markdown library isn't something that ships with Python, you will need to install it using pip using 'pip install markdown'. Now if you use 'article_object.content_html' it will give you an HTML representation of Markdown contents available via 'article_object.content'.

The '@property' before the method definition tells Python that the value of this function should be accessible as a property; so you can access it via 'article_object.content_html' instead of 'article_object.content_html()'. This marks no practical difference in our case, but hey you got to learn something new.

DJANGO URL ROUTING

Learning Django's URL routing mechanism from the inside out will let you create a URL structure that clean, neat and easy to decipher

Now that we have our models set up, let's start getting something showing in our browser. If you've ever used PHP you might have started with learning how URLs are mapped to files on the filesystem. So a URL like 'mysite.com/about/myproduct.php' would be rendered by an actual file called 'myproduct.php' in the 'about' folder of the webserver's directory on the 'mysite.com' server.

Most web frameworks including PHP ones don't do that though, rather they usually have a single file, or code that decides what code to call for what URL. In DJango this function is fulfilled by the 'urls.py' file. The default layout for our project (if you followed all the instructions and used the same names as us) will have this file placed in the `mycms` folder in out project folder.

The 'urls.py' file, as we explained before, is responsible for connecting the URL a user visits on your website to some code in your project that generates and returns the content to display on that page.

If you open this file right now you will notice that it already has some code in it, and has comments that describe how this file can be used.

You will notice a list variable called `urlpatterns` that has a single element:

```
url(r'^admin/', include(admin.site.urls)),
```

This line of code tells Django that URLs beginning with 'admin/' should be passed along to the admin framework included in Django. Let's leave this alone for now and look at the general syntax for a defining URL mappings.

The way you map a URL to code is to add additional 'url' entries as above that provide a pattern to match, and a view function or class to handle that pattern. So 'url(pattern, view)' to match a pattern to a view.

A simple example of a pattern would be as follows:

```
url(r'^test/', views.test_view),
```

You could add this line to the 'urlpatterns' list and it will instruct Django to call the 'test_view' function in our views file and display its response in the browser.

Currently of course this function does not exist. Additionally you will need to use the import statement to import the views package if you want to be able to use it as above. So let's do that.

Creating a Simple View for Testing

Let's head over to our views.py files in the `cms` app folder and type in it the following code:

```
from django.http import HttpResponse

def test_view(request):
    return HttpResponse(
        'This URL is properly mapped.'
    )
```

We'll leave out how this works exactly to the chapter about views. For now it's sufficient to know that we are returning a simple text response from this view that says 'This URL is properly mapped.' You can change this text to whatever you want and it will show up in your browser. If you turn this into HTML text, the browser will render it like any other web page.

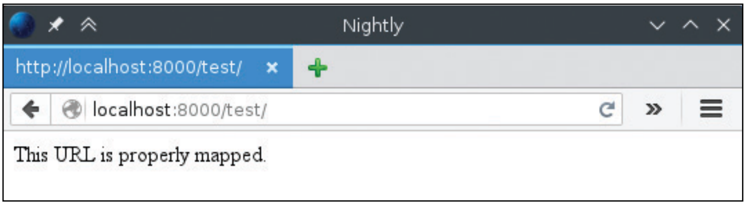
Right now though you need to first properly import the views file in the urls file to have this work. As such replace the contents of the 'urls.py' file with the following:

```
from django.conf.urls import include, url
from django.contrib import admin
from cms import views
```

```
• urlpatterns = [
•     url(r'^admin/', include(admin.site.urls)),
•     url(r'^test/', views.test _ view)
• ]
```

You'll notice that the only way this differs from the code you already have is in the addition of the pattern for the test URL and the 'from cms import views' import statement.

Now rerun the 'python3 manage.py runserver' command (or the Windows alternative) and point your browser to '*http://localhost:8000/test/*' and you will see the message we typed in the 'test_view' function show up here. Try playing around with the message and using HTML code in it to see what happens.



This URL is properly mapped.

We've successfully managed to route the /test URL to our own code

Also try adding more patterns; just copy that line and paste it a couple of times with the first bit altered. Let them all point to the same function for now. You will now see that those URLs are also accessible and produce the same output!

Now try making copies of the 'test_view' function, change the functions' names and change the responses as well. Now have different URL patterns point to different functions, and see that different URLs will produce different responses. This is to a large extent how most of our CMS will work. We will create a pattern in the urls file, link it to a function that produces the content for that content, and voila we have a CMS.

Matching With Patterns

You may be wondering though, that if this is how matching URLs works, you will have to create a separate 'url' entry and view function for each possible date, category, tag, and even article. Don't worry that is not the case!

There is a reason we always call the first parameter inside `url` the pattern, and it's because it is supposed to match a wide range of URLs.

The URL matcher supports a popular syntax for pattern matching called RegEx (or Regular Expressions) that can be used to describe all kinds of patterns and text.

In fact even the simple example ('^test/') we used is a RegEx pattern that can match a whole range of URLs! First, notice that it begins with the caret symbol '^' and that is noware to be seen in our URL. In RegEx this symbol stands for the starting of the text to be matched. There is also a symbol for the end of the text, and that is represented by the dollar symbol '$'. Note that we have not used that symbol in our pattern.

So the pattern '^test/' means: Match any URL path that starts with 'test/'. So it need only start with 'test/', you can enter anything else after that and it will still match that pattern and display that same message! For instance the URL '*http://localhost:8000/test/abc123'* will match this patten and return the same message. Go ahead, try it.

Note that these patterns only match the URL path after the host name. So in the URL '*http://localhost:8000/some/path/'* the beginning '*http://localhost:8000/'* bit is never part of the pattern. So in the future we will leave out this bit of the URL for brevity since it is inconsequential to URL matching.

Now if you add a '$' at the end of this patterns and make it '^test/$' it will stop matching anything other than just '*http://localhost:8000/test/'.* This is because the pattern now means: Match text that contains 'test/' between its starting and ending. This of course is only true for that one URL.

On the other hand if you remove even the '^' in the beginning, it will match any URL that contains ` test/' anywhere in it. So it will match all of the following:

- `abctest/`
- `123test/`
- `admin/test/`
- `admin/edit/test/`

The last two of the above cases should be especially worrying since they normally would lead to your admin panel. This matching happens in order, so if a URL matches multiple patterns, the first view function is the one that will be called, and its response retured to the user.

RegEx supports matching some very complex patterns, but we wont need a lot of that here. What we will need is the ability to match year / month / day numbers, category / tag / article / gallery / image names etc. All of this is easily possible with RegEx.

If you want to match a multiple characters, you can specify them as a range or list within square brackets. For instance, the pattern `^test/[0-9]$` will match any URL that starts with 'test/' followed by a single number from 0 to 9. It will not match any URL that ends with an alphabet or even a two-digit number. If we just wanted a match for the digits 2, 3, and 5, we could instead use [235]. For alphabets we can similarly list them or use ranges. For example [a-q] or [acf].

If instead we wanted to match any arbitrary sequence of numbers instead of a single digit, we could use the pattern '^test/[0-9]+$'. Here the plus sign '+' means one or more occurrences, so this will match any URL that is `test/' followed by any one or more digit number. If you use an asterisk '*' instead of the plus sign, that will include the case where there is no digit present.

If we want to be able to list articles by date — like in a blog we need some way to match a year number, month number and day number. In this case the year number can't be of arbitrary length, it has to be four digits. Likewise the month and day numbers have to be of two digits each. The way to specify this pattern is to use curly braces enclosing the number of times the pattern needs to repeat within them, instead of a plus or asterisk.

So to match a year you would use: '^archive/[0-9]{4}/$'. This would match any URL starting with 'archive/' followed by a four-digit number, and ending with a '/'. Likewise to match a year and month we would use: '^archive/[0-9]{4}/[0-9]{2}/$'

We can repeat the second bit that matches the month number once again to have it match days as well. If we have a situation where we want to match between, say, 3 and 6 digits, the pattern to use is '[0-9]{3,6}'.

Finally let's look at how to match for slugs that are used for articles, categories etc. But first let's understand what a slug is.

Often times titles for articles and other pieces of content can use special characters and spaces. Those aren't great for URLs as they need to be escaped and leave you with odd looking URLs that aren't good to look at or understand.

For instance let's look at an article titled "This is an Article Title: Something's Happened - Part 2" — quite the popular one as sequels go. If we include this title in the URL, it will appear as:

- `This%20is%20an%20Article%20Title%3A%20Something%27s%20Happened%20-%20Part%202`

As we said, it's not pretty! What we do instead is that we create a cleaner version of this title that is URL friendly while still being decipherable. This is called a slug. If we set things up correctly Django will make one for us.

For this same title the slug Django will produce will be:

- `'this-is-an-article-title-somethings-happened-part-2'`

As you can see this is a lot cleaner, and easier to understand. When this article is published it will appear as:

http://some.website/articles/this-is-an-article-title-somethings-happened-part-2

There are many ways we can match this pattern. An obvious one from what we have learned so far would be to match for lower case characters, numbers and the hyphen '-'. This would be represented as [a-z0-9-], there is also a shorter form [\w-]. There are many such short forms in RegEx, '\w' stands for all lower and upper case characters, all numbers and the underscore, or [a-zA-Z0-9_].

To match an article slug we would now use the following pattern: '^articles/[\w-]+/$'

Extracting Parameters from URLs

Well, all this is great; it allows us to match a range of URLs, but simply matching isn't enough in the end, we also want to be able to change the content of the page based on the matched bits of the URL.

So if we match a URL like 'archive/2014/04/' we want the view to know that we matched 2014, and 04 so it can filter the articles for that month of that year and display them.

This is in fact very very simple, as simple as simply placing brackets '()' around the bits of the URL that we want the view function to receive! So the pattern for capturing the year, month and day in the URL becomes:

- `^archive/([0-9]{4})/([0-9]{2})/([0-9]{2})/$`

This will capture the year, month and day separately from this URL and pass those along to the view function where these parameters can be used to fetch the appropriate articles. The old 'test_view' view function we wrote earlier wont work for this, since the view function needs to be capable of receiving these parameters as well. It's as simple as adding additional parameters to the function declaration, one for each value captured. We'll look into this in more detail in the next chapter.

Regular Expressions also allow you to give a name for the data you are capturing; for instance the pattern for capturing the year could be written as:

- `^archive/(?P<year>[0-9]{4})/$`

You'll notice that we have added '?P<year>' before the pattern inside the capturing bracket. This explicitly names this capture as the year. There isn't a huge advantage in doing this in most cases other than being explicit. Explicitly capturing naming your slugs as 'slug' can be advantageous though when dealing with class-based views (which we will cover later in this text). As such we will leave pattern names except when capturing slugs.

Our URL Configuration

We've discussed the URL structure we want to use in the previous chapter, so now let's get on to implementing it!

Let's look at all the pieces of content and see how their URLs will be implemented.

For articles we need to have a main 'articles/' URL to list all articles, and a 'articles/<article-slug>' URL for individual articles:

- `^articles/$`
- `^articles/(?P<slug>[\w-]+)$`

We can also make it possible to browse articles by date, we've already seen these before, but here they are again:

- `^archive/([0-9]{4})/$`
- `^archive/([0-9]{4})/([0-9]{2})/$`
- `^archive/([0-9]{4})/([0-9]{2})/([0-9]{2})/$`

Like for articles, we need to have a 'categories/' page to list all categories and a 'categories/<category-name>' page for each category. We need similar pages for tags, and galleries.

- `^categories/$`
- `^categories/(?P<slug>[\w-]+)$`
- `^tags/$`
- `^tags/(?P<slug>[\w-]+)$`
- `^galleries/$`
- `^galleries/(?P<slug>[\w-]+)$`

Images don't have a listing page, but they still need a page for each image.

- `^image/(?P<slug>[\w-]+)$`

Finally, we have Authors. Now Authors don't have a slug! However each author does have a unique username and serial ID. We can use one of these. We will use the username since the user's serial ID is an internal piece of data that the user shouldn't need to be aware of, and should preferably be kept hidden. How we map the slug here to a username is something you will learn in the next chapter.

- `^authors/$`
- `^authors/(?P<slug>[\w-]+)$`

Those are all the patterns you will need for this CMS. Since we haven't written the views yet, you can just link them to dummy views for now.

Since we don't have a proper server like Nginx or Apache set up to serve our website and files, we need to add an additional entry to our `urlpatterns` setting that tells Django to serve any media files we upload. This URL entry is as follows:

```
url(r'^media/(?P<path>.*)$',
    'django.views.static.serve', {
        'document_root': settings.MEDIA_ROOT,
        'show_indexes': True
    }),
```

This tells Django to serve our media files located in the MEDIA_ROOT directory configured in the settings file from the URL 'media/'. Setting `show_indexes` to True also allows us to directly visit `media/` and get a list of files being served from there. This should be used during development only. Since this uses a settings file from settings, you will also need to add `from mycms import settings` at the top of the `urls.py` file.

DJANGO VIEWS

Django view's do the bulk of the work in implementing the actual functionality of the website. The more complex a website, likely the more complex its views

In the previous chapters we were very roughly introduced to a very basic view. That simple view that we — in our inestimable hubris — named 'test_view' it did the job for the time, it returned a simple response for whatever URL that linked to it. It's the view we deserved, but not the one we need right now.

The view we need right now is one that can display an article on the page. Django supports two ways of doing this. One is to use function-based views, where a URL is mapped to a function, the function receives the parameters from the URL, uses those parameters to generate HTML code to display to the user, and then returns that code as a response.

The other way is to use class-based views. These have been around for a long time, but initially Django only supported function-based views. The advantage of class-based views is that Django ships with many generic view classes that can handle a lot of common use cases such as displaying a list of items, or displaying details about a single item etc.

We'll first see how we can make this simple article details view using both a function and a class so we can compare how both work.

Comparing Function-Based and Class-Based Views

We'll begin with a function-based view for the article detail view. Remember that this function will receive a parameter, the slug of the article. We need to take this slug, and use it to fetch the article with that slug from the database. From the 'Django Models' chapter we know how to fetch a single item from the database:

```
Article.objects.get(slug=article _ slug)
```

We'll use this in our view function to get the correct article, fill in details from this article into an HTML string, then return this code in an *HttpResponse* and Bob's your uncle! Although given the demographic our our readership your uncle's probably Sanjay.

Here is the code for our view:

```
def article _ detail(request, slug):
    article = Article.objects.get(slug=slug)
    html = """
    <html>
    <head><title>{title}</title></head>
    <body>
    <h1>{title}</h1>
    <p>Author: {author}</p>
    {content}
    </body>
    </html>
    """
    response = html.format(
        title=article.title,
        author=article.author.username,
        category=article.category.name,
        content=article.content)
    return HttpResponse(response)
```

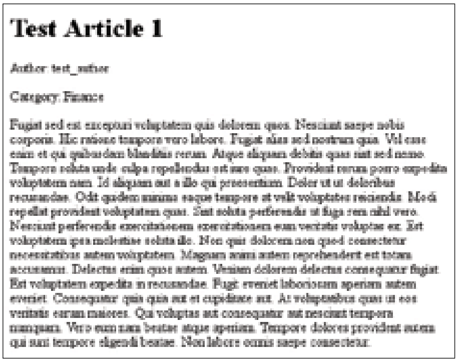
You will also need to map this view to the correct URL in the `urls.py` file. The correct entry mapped to this view will look like the following:

```
url(r'^articles/([\w-]+)$', views.article _ detail,
    name='articles-details')
```

You'll note that we've added another bit to our URL configuration, a name. While it is not necessary to supply a name for your URL patterns, supplying one makes it easy to reverse a URL match, that is, to get a URL for an article given its ID or slug.

Now visit '*http://localhost:8000/articles/test-article-1*' and you will be greeted by a bare-bones page that shows the article title, author, category and content.

When you learn about Django templates, you will be able to simplify this code a lot by moving the HTML bits of the code to an external file. To give you an idea, if this HTML code is converted to Django's template format and moved to a template file called 'article_detail.html' in the templates directory, this function is reduced to the following:



Test Article 1

Author: test_author

Category: Finance

Fugiat sed est excepturi voluptatem quis dolorem quos. Nesciunt saepe nobis corporis. Hic ratione tempora vero labore. Fugiat alias sed nostrum quia. Vel esse enim et qui quibusdam blanditiis rerum. Atque aliquam debitis quas sint sed nemo. Tempora soluta unde culpa repellendus est iure quas. Provident rerum porro expedita voluptatem nam. Id aliquam aut a illo qui praesentium. Dolor ut ut doloribus recusandae. Odit quidem minima eaque tempore et velit voluptates reiciendis. Modi repellat provident voluptatem quas. Sunt soluta perferendis ut fuga rem nihil vero. Nesciunt perferendis exercitationem exercitationem eum veritatis voluptas ex. Est voluptatem ipsa molestiae soluta illo. Non quis dolorem non quod consectetur necessitatibus autem voluptatem. Magnam animi autem reprehenderit est totam accusamus. Delectus enim quos autem. Veniam dolorem delectus consequatur fugiat. Est voluptatem expedita in recusandae. Fugit eveniet laboriosam aperiam autem eveniet. Consequatur quis quia aut et cupiditate aut. At voluptatibus quas ut eos veritatis earum maiores. Qui voluptas aut consequatur aut nesciunt tempora numquam. Vero eum iusto beatae atque aperiam. Tempore dolores provident autem qui sunt tempore eligendi beatae. Non labore omnis saepe consectetur.

A bare-bone page showing an article's content

```
def article_detail(request, article_slug):
    article = Article.objects.get(slug=article_slug)
    context = {'article': article}
    return render(request, 'article_detail.html',
                  context)
```

That looks a lot better!

Now let's see how we'd do the same with class based views. First we will implement using classes but with using an external template.

```
from django.views.generic import View

class ArticleDetail(View):
    html = """
           <html>
           <head><title>{title}</title></head>
           <body>
           <h1>{title}</h1>
           <p>Author: {author}</p>
           <p>Category: {category}</p>
           {content}
           </body>
           </html>
           """
    def get(self, request, article_slug):
        article = Article.objects.get(slug=article_slug)
        response = self.html.format(
            title=article.title,
            author=article.author.username,
            category=article.category.name,
            content=article.content)
        return HttpResponse(response)
```

The way to map this to a URL in the 'urls.py' file is a little different. Where you used 'views.article_detail' in the case of a function-based view, you will now need to use 'views.ArticleDetail.as_view()'

Depending on your proclivity toward object-oriented programming you might find this better / cleaner or worse than the function-based approach. Wait till you see how class-based views work with templates though, as that might change your mind.

With a template placed in the correct place, this class-based view reduces to:

```
from django.views.generic import DetailView

class ArticleDetail(DetailView):
    model = Article
```

This is possible in part because we named the slugs we capture from the URL. Django knows how to handle this case automatically. This also requires that the template files be placed in the correct directory, and have the correct name. In this case you would need to name the file 'article_detail.html' and place it in a folder called 'cms' (same as the name of our app) in the templates folder. Of course all of this can be changed and configured if you want, but if you follow the conventions, it can be quite rewarding.

All of this will become a lot clearer when we study templates in more detail. Before that though, we will look deeper into class-based views, they seem to have some interesting things going on.

Generic Class-Based Views

A lot of the work in rendering different web pages is actually quite repetitive. Whether it is a list of articles or a list of galleries, we need to perform a similar set of tasks. First we need to fetch a list of items, and then we need to pass them to the template to build our webpage, and then we need to return this to the user as a response that will be rendered in the browser.

What's different between and Article, and a Gallery in this case is the name of the model, and the template used to render this list of items. So to ease the task of creating these common kinds of views, Django includes what it calls Class-Based Generic views.

Generic views called ListView and DetailView (there are others) are available for developers to base their class-based views on so that they can focus on the bits that are different, such as the templates.

We already saw the DetailView in action as it rendered an article page using just two lines of code. If we wanted to use the ListView generic view for rendering our article list page, the procedure is exactly the same. The only code we will need is the following:

```
class ArticleList(ListView):
    model = Article
```

This view will automatically fetch a list of articles, load the template from the conventional location (cms/articles_list.html derived from <app name>/<model name>_list.html) supply it the list of articles to render it and then return that data to display in the browser. You could just make a copy of the above and put in Gallery instead of Article and have a functioning Gallery view!

You may be wondering, what's left to do now? Class-based generic views have solved all our problems, so let's just move along to the next chapter. Unfortunately, we have cheated till now and you will soon understand why the above code can't work for the article list, or even the article.

We have conveniently ignored till now the fact the we only want to display a list of published articles. The views above have no way of knowing that and end up showing all articles, even the unpublished ones. This is of course bad, but the good news is that the solution is very simple and is as small as the above code sample.

Our solution to only display articles have are published is as follows:

```
class ArticleList(ListView):
    queryset = Article.objects.filter(
        publish _ status=True
```

Notice that instead of specifying a model, we are directly specifying the list of objects to use in for this ListView. This is still only two lines of code, works the same way as before, with the same template.

We can similarly use a queryset with our ArticleDetail view. If we continue to use the old ArticleDetail class that specifies a model instead of a queryset, it will be possible for people to open an unpublished article if they know the slug even if the article is unlisted. It will work like unlisted videos on YouTube, where knowing the URL lets you open the video but it is not otherwise viewable. This might be what you want in your CMS.

By supplying a filtered queryset to the ArticleDetail view we limit the articles it is allowed to show to the ones that are filtered. Any article not in the list of filtered articles might as well not exist as far as the view is concerned. Again, neither approach wrong, both have their uses.

Since this code is very simple, we won't bother discussing all the different views we need to have. Rather we will just discuss some of the more interesting cases that test the limits of generic views.

The first view that we will have trouble implementing with the generic views we have discussed so far is the home page itself. We would want this page to list a few different types of content, and generic views only support pulling details from a single model per view. For this page we could either use a custom class-based view, or a function-based view.

The other view that might cause trouble with these views is the tag list page. We would like to render this as a tag cloud, and in order to do that we need extra details, such as the number of times each tag is used, and this isn't something that is available by default when listing tags.

The author detail view isn't special at all, except that we want to be able to link to authors by their username rather than a numeric ID or slug (since users don't have a slug). A user's username can be used as a slug, but this wont happen automatically, we will need to configure the ArticleDetail class to perform this conversion.

Finally, our gallery detail view which will need to support multiple layouts. While this is an interesting case, it is entirely possible to do with a standard DetailView by using some basic logic in the template rather than in the view.

Since these are the troubling cases, we'll look at these in more detail.

The Author Deail View

Let's star with the simplest of all cases, the AuthorDetail view which is just a matter of configuring the view to look for the slug information in the right place.

```
class AuthorDetail(DetailView):
    model = User

    def get _ slug _ field(self):
        return 'username'
```

This is all you need to do for the author detail page to start working! Django calls the 'get_slug_field' method here to get the name of the slug field in our model. Since the user model doesn't have such a thing we just help Django out a little in looking for the right thing.

The Tag Cloud View

In order to size different tags based on how many articles they appear in, we need to construct a query that includes this data in before sending it to the template. So in the end this particular pickle is all about finding the right queryset to get the desired results.

Django's ORM has exactly such a feature, called annotations, but it isn't something we have covered in the 'Django Models' chapter. The annotations feature allows us to add additional fields to each item in a queryset that are aggregated values calculated from fields of that model. For instance, if we had a model that had two numeric fields, 'value1' and 'value2', we could use annotations while constructing a query to add in another field 'value_diff' that is the difference between these fields.

Our case is a little different though, we want to add to each tag a count of the number of articles that have that tag applied to it. The query for this would go as follows:

- `from django.db.models import Count`

- `Tag.objects.annotate(num _ articles=Count('article'))`

This will return a list of tags where each tag has an additional field called 'num_articles' which is the number of articles that have that tag. We can now use this as the queryset of the TagList view and use the 'article_count' value in the template to change the font sizes of the different tags.

Gallery Detail View

We wanted to be fancy and give users the ability to select one of three layouts for our gallery page. To actually implement this we have many different options, but they all start with one decision: do we want to have the view select which template to render, or do we want to have a single template that has all three layouts (within the same file or split out) and selects the layout in the template.

Our preference is to keep as little logic and code in the templates as possible, so our preference will be to have three different templates and have the view select which template to render based on the layout selected by the user.

As we know, the ListView generic view automatically uses the correct template to render based on the name of the application and the name of the model. However that doesn't suffice in this case because it requires us to select a template based on a setting stored in the model.

We also mentioned that the location and name of this template file is customisable if we want to override it. All we need to do is add a 'template_name = 'path/to/template.html'' attribute like we added a 'model = Article' or 'queryset = Article.objects.all()'. This doesn't satisfy our needs either since our template name needs to change based on the layout value of the gallery item.

What we need to do here is to override the 'get_template_name' method of our GalleryDetail view class to return the correct template based on the value of the currently displayed object. Here is how we go about doing that:

```
class GalleryDetail(DetailView):
    model = Gallery

    def get _ template _ names(self):
        if self.object.layout == 'HORI':
            return (
                'cms/gallery _ detail _ horizontal.html',
            )
        elif self.object.layout == 'VERT':
            return (
                'cms/gallery _ detail _ vertical.html',
            )
        else:
            return (
                'cms/gallery _ detail _ links.html',
            )
```

The Home View

If you go back to the 'Designing our CMS' chapter where we discussed our home page layout, we mentioned that we wanted our home page / front page to display a list of the top two categories, the latest ten articles (two in full and the rest as links) and other links such as a link to all the list pages for articles, categories, tags, galleries, and authors.

Some of these are just static links but at the very least this page needs to display the top category names and ten articles. For this it needs to be provided the latest ten articles, and a list of the top categories, both of which come from different models.

With what we have learnt so far getting the latest ten published articles shouldn't be difficult. We've already set articles to be ordered by creation date. Getting the top two categories can be a little trickier. First one all we need to know how many articles each category has which should be easy now that we know about annotations. Let's first annotate a Category with the number of articles in each category:

```
Category.objects.annotate(
    num _ articles=Count('article'))
```

We can now use the 'order_by' method to sort these categories by this newly created 'num_articles' field! After that we can use the standard Pythonic approach to getting a slice of an array to get the first two categories.

```
categories = Category.objects.annotate(
        num _ articles=Count('article')
    ).order _ by('-num _ articles')[:2]
```

You'll notice that we passed '-num_articles' to 'order_by' instead of 'num_articles' (there is an extra minus sign), this is because we want it ordered in reverse, or in descending order so that the top two categories are the first two results.

Now that we know how to get both the latest ten articles and the top two categories, we can combine them into a single context and pass it to the 'render' function to render our home template.

Here is what that looks like:

```
def home(request):
    articles = Article.objects.filter(
        publish _ status=True)[:10]
    categories = Category.objects.annotate(
        num _ articles=Count('article')).order _ by(
            '-num _ articles')[:2]
    context = {
        'articles': articles,
        'categories': categories
    }
    return render(request, 'cms/home.html', context)
```

That was simpler than it sounded wasn't it. Of course right now you don't know what is going on inside this `home.html` template file, but it isn't long till you find out.

DJANGO TEMPLATES

While you can use many different template engines with Django, the default one that ships with is is nothing to scoff at!

One of the most important tasks a web application has to perform is to convert the raw data it retrieves from the database — or any other place really — into a visually pleasant HTML representation that the user would actually want to see.

There is no shortage of template engines that are designed to do exactly that. There are template engines available for most programming languages and they can generate a variety of data, not just HTML code. Template engines often support some form of template language / syntax that tells it what kind of data should go where in the template. After that all you need to do is provide the template engine a template, and the data it needs to plug into the template and we get the output we want — hopefully.

Django happens to ship with its own template engine that is designed for the web, and with the generation of HTML code in mind. The template language Django uses allows can include all kinds of logic, looping over items in a list, or checking if some condition is true or false, however what it does not support is the inclusion of any kind of Python code in the template itself. Any complex logic needs to stay in your Python code.

If for some reason you don't like how Django's template language works, you can plug in another template backend, or just import some other Python library to handle this bit for you. Of course in this case it might become harder for you to integrate with other parts of Django that use its own template language, such as the admin panel.

So without further ado, let's introduce the syntax of Django's template language, the final piece in our understanding of Django.

The Django Template Language

The first thing to know about Django's temple language is that it doesn't know, or care if the output it is generating is actually HTML or not, all it sees is text. While this means that you can generate non-HTML, but text based output using these templates, it also means that you are fully responsible for ensuring that you are generating valid HTML, which can be harder when you have non-HTML template code in the same file.

Another thing you should know before diving in is, where do you keep these templates anyway? Since templates are associated your 'cms' app, you will need to place them in a folder called templates in your 'cms' app directory. Here too it is important to note the conventions Django applications tend to use. Since out application is named 'cms' we need to create a directory called 'cms' within the templates directory for Django to automatically find some of our templates.

In the previous chapter, before we showed how to use templates, we just kept our HTML code in a Python string; keep that code in mind while we present to you the Django template code to generate the same output:

```
<html>
<head><title>{{ article.title }}</title></head>
<body>
<h1>{{ article.title }}</h1>
<p>Author: {{ article.author }}</p>
<p>Category: {{ article.category }}</p>
{{ article.content }}
</body>
</html>
```

Notice how similar this is to the code we originally used! In fact the major difference seems to be that we are using double curly braces instead of single ones. If you flip back the page with the original code, you will see that while generating a response from this template we passed it a context object that had an attribute 'article' set to the article to display, in this template we are accessing the details from that article object.

Now let's go over the syntax of Django's template language. Also remember that it's possible to write your own code that extends Django's template syntax.

Displaying Data

Displaying stuff in the right place is one of the most important things you probably want to do in a template, and the syntax for that is quite simple, as you saw in the previous example. All you need to do to print a variable is to enclose the name of the variable inside double curly braces. As you saw in the previous example, you can access the attributes of an object as well, however it is also possible to access elements of an array as follows:

```
{{ array.1 }}
```

Essentially for anything after the period it will check to see if it is an attribute or function or array index. In case it is a function it will call the function (if it supports being called without any arguments) and print the value.

Conditionals

There are cases where you might want to display different messages based on some or the other condition, for that we have the `if` block that can be used as follows:

```
{% if article.publish _ status %}
    Published
{% else %}
    Not Published
{% endif %}
```

How this works should be pretty obvious given the above example. Since publish_status is already a binary field this works out for us, however this can also be useful to check if a property has been set. For instance since our featured image is optional, we can use code like the above to check if a featured image is set, and to hide the image block if there is no image to display.

The 'if' block also also obviously evaluate conditionals like the following:

```
{% if article.tags.count > 2 %}
```

The article has more than two tags.

```
{% endif %}
```

Loops

Speaking of tags, articles have many of those, so we need to be able to display lists of items. Also we will need to do this on pages that list published articles categories etc. The Django template language has a for loop block that can iterate over a lists of items. So to print a list of all the articles tags, you could do the following:

```
{% for tag in article.tags.all %}
    {{ tag.name }}
{% empty %}
    No tags
{% endfor %}
```

Note that here we have also supplied a message to display if there are no items in the list, this is optional.

Another thing we may want to do is to keep track of the loop counter, and possibly display it or use it in some other way. This counter can be accessed via 'forloop.counter' if you want a counter starting from 1 or `forloop.counter0` if you want a counter starting at 0. In either case you can print this counter by placing it inside double curly braces as usual.

URL

Another incredibly important thing we'd want to do is to link between different kinds of data in our database. We'd want all articles in the article list to be linked to articles, and all tags on an article page to go to that tag's page etc. For this we need to be able to get the URL of these objects to include within a link tag in our code.

If you'll recall we mentioned that you should provide a name for all your views, and this is the reason why, if you know the name of a URL pattern, you can generate a URL for it given all the parameters. For instance to generate the URL to a particular article we can do the following:

```
{% url 'articles-detail' article.slug %}
```

Here we are assuming that the name of the URL pattern for the article details page is 'articles-detail'. You could use this to generate a link with the following code:

```
<a href="{% url 'articles-detail' article.slug %}">
  {{ article.title }}
</a>
```

Filters

Filters are another important part of Django templates. They allow us to process variables before being displayed or used in our templates. For example, if you want to convert some string to lower case before outputting it in your final HTML you can use the `lower` filter as follows:

- `{{ tag.name|lower }}`

Here are some other useful filters:

- lower: convert to lower case
- upper: convert to upper case
- date: format a date value into a human-readable format
- escape: escapes HTML code so it is displayed as code rather than being rendered by the browser
- filesizeformat: converts a number like 1024 into a human readable file size like '1 KB'
- join: joins the contents of a list into a single string. Use as {{ list|join:", " }} to separate elements of the list by commas
- length: returns the length of a list
- linenumbers: prints text with line numbers
- pluralize: switched between singular / plural depending on whether the value is 1 or more. So 'tag{{ tags.count|pluralze }}' will output 'tag' if tags.count is 1, and 'tags' otherwise. You can specify the pluralisation scheme as well in case pluralising the term used is more complex than just adding an 's'.
- safe: By default Django will escape any text so that its HTML tags are printed on page rather than being rendered. This ensure that any comments of other user-generated content can't affect the layout of the page or perform actions that the website designer did you expect. Of course we might occasionally want to actually render some HTML code on screen, such as for the article contents. In these cases we can use the `safe` filter to mark the text as safe for rendering.
- slugify: converts a string into a slug
- striptags: strips the tags from HTML code. Very useful for comments, or other places where user input is used.
- time: format a time value into a human-readable format
- timesince: prints the time since the supplied date. So it will convert an unfriendly date time into something simpler like '3 hours ago'
- truncatewords / truncatewords_html: Truncates plain text / HTML to the specified number of words.
- urlencode: urlencodes a string
- urlize: converts a URL or email address into an actual link
- wordcount: calculates the wordcount of the text

There are even more filters than the ones listed above, and yet more can be added by installing packages that provide filters. A package with some extra tags and filters called 'django.contrib.humanize' is included with Django. It has useful filters such as 'intcomma' that add commas into a long number, and 'intword' that converts a number like 2400000 to the string '2.4 million'.

It is also possible to chain multiple filters together in which case they will be applied in the order that they are listed.

Extending Templates

One of the most powerful aspects of Django templates is something we haven't covered till now at all, and that is the ability to extend one template from another, and the ability to include bits from other template files. Templates support a form of inheritance than significantly reduces the amount of code you need to repeat.

The way template inheritance / extension works in the Django template language is that first create a base template that has the HTML framework of your website. This base template will probably have a basic HTML file skeleton with the root 'html' tag, and the 'head', 'title' and 'body' tags. You might also include any scripts and stylesheets that are applicable throughout your website. If you want you can even fill in some content, such as a site menu, a site header and footer, sidebars etc.

In this base page you also include special Django template elements called 'blocks' that serve as points of extension. These blocks, (and only these blocks) can be then overridden by any child templates. Child templates need not override all the blocks, and any blocks they don't override will show the default content defined by the base template.

Let's look at an example of a base template (site_base.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>
    {% block title %}My CMS{% endblock %}
  </title>
</head>
<body>
<aside id="sidebar">
  {% block sidebar %}
    <menu>
```

```
•       <li><a href="/articles/">
•         All Articles
•       </a></li>
•       <li><a href="/categories/">
•          Browse Categories
•       </a></li>
•     </menu>
•   {% endblock %}
• </aside>
• <main role="main">
•   {% block content %}{% endblock %}
• </main>
• </body>
• </html>
```

The above template will serve its purpose for our entire site, since all pages will need to have a sidebar with links, and all pages will need to have a 'main' tag for the main content. This also template still encloses the contents of the sidebar in a block called 'sidebar' allowing us to optionally override its contents if need be. This isn't the final code for the base template, as in our final page we'd want to include some CSS and a some more links in our sidebar.

Now let's look at how the article detail template can extend from this (cms/article_detail.html):

```
• {% extends 'site_base.html' %}
• {% block title %}
•   {{ article.title }}
• {% endblock %}
• {% block content %}
•     <h1>{{ article.title }}</h1>
•     <p>Author: {{ article.author }}</p>
•     <p>Category: {{ article.category }}</p>
•     {{ article.content }}
• {% endblock %}
```

Here the first 'extends' tag tells Django that this template is a child of the 'site_base.html' template. After that all we need to specify are the blocks that we want to change. In our case we want to set the title of the page to the title of the article, and we want to replace the 'content' block with the article details. We haven't overridden the sidebar, and as such that will show the content we defined in the base template.

We needn't stop extending here! Even in this child template we can create further blocks within these blocks and override them in templates that extend from this one!

Since we want to support three different layouts for our gallery, we can take advantage of this while coding our gallery templates. We can have one base gallery template that extends the site_base template and then three templates that extend from the base gallery template to implement the three layouts.

There is actually another way to go about this using the 'include' tag of the Django template language. Let's have a look at the include tag next.

Including one Template in Another

Where template inheritance allows use to reuse the same base template across many different sections of our website, includes provide another form of code reuse that works better for short bits of code.

When you include one template in another using the 'include' tag, the included template is rendered by Django and its contents are inserted directly in the template in the position of the 'include' tag. Here is how you can use the include tag in your code:

```
{% include 'template_name.html' %}
```

Imagine reusing a widget like the tag cloud. You might want to use this is many different places in the website, so it might not make sense to have it included in a fixed location in the base template. We might want the tag cloud in the sidebar on the article detail page, but in the main content view on the tag list page. Without includes this would require us to have multiple copies of the code rendering the tag cloud.

Using the include tag in this case solves our problem by letting us put this tag cloud code in an external template file and then including it in wherever needed. We can even pass it some context data as shown in the following example:

```
{% include 'tag_cloud.html' with tags=article.tags.all
%}
```

You can use this to simplify creating and adding all kinds of widgets. Wherever you feel like you are typing the same HTML and Django template code again in another file, you should consider placing that code in an external template file and including it wherever needed instead.

Now that we have a sufficient understanding of Django templates, let's go about making templates for all our pages.

Coding our Templates

We will now look at the templates we need one by one and discuss any pieces of code where confusion is likely.

Let's start with the `site_base.html` template since the one we looked at before was incomplete.

site_base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>{% block title %}My
  CMS{% endblock %}</title>
<link rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/founda-
tion/5.5.2/css/foundation.min.css">
</head>
<body class="row">
<aside id="sidebar" class="small-3 columns">
{% block sidebar %}
  <menu class="side-nav">
  <li><a href="/">Home</a></li>
  <li><a href="/articles/">All Articles</a></li>
  <li><a href="/categories/">Browse
    Categories</a></li>
  <li><a href="/galleries/">Browse Galleries</a>
  </li>
  <li><a href="/authors/">Browse Authors</a>
  </li>
  <li><a href="/tags/">Browse by Tags</a></li>
  </menu>
{% endblock %}
</aside>
<main role="main" class="small-9 columns">
  {% block content %}{% endblock %}
</main>
</body>
</html>
```

You'll notice we've expanded on this template quite a bit. First of all we are now including the Foundation CSS grid framework to make the website at least functionally presentable. We won't go into the details of how it works or how to use it, but suffice to now that by adding the classes we have above ('class="row"' it the body, etc) we are ensuring that the sidebar takes up 3 columns of space to the left, and the main content takes up the 9 columns to the right. We wont use any additional CSS code, but feel free to add your own to make the website look better.

Other than that you will notice that we have added all the links to the sidebar now.

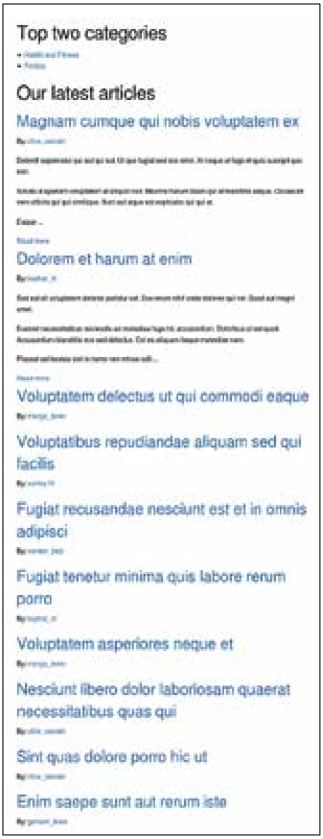
cms/home.html

```
• {% extends 'site _ base.html' %}
• {% block title %}
•   Welcome to MyCMS
• {% endblock %}
• {% block content %}
•   <section>
•   <h1>Top two categories</h1>
•   <ul>
•   {% for category in categories %}
•     <li>
•       <a href="{% url 'categories-detail' category.slug
%}">
•         {{ category.name }}
•       </a>
•     </li>
•   {% endfor %}
•   </ul>
•   </section>
•   <section>
•   <h1>Our latest articles</h1>
•   {% for article in articles %}
•     <article>
•     <header>
•     <h2>
•       <a href="{% url 'articles-detail' article.slug %}">
•         {{ article.title }}
•       </a>
•     </h2>
•     <p>
•     By:
•     <a href="{% url 'authors-detail' article.author.user-
name %}">
•           {{ article.author }}
•       </a>
•       </p>
•       </header>
•       {% if forloop.counter <= 2 %}
•         <section>
•           {{ article.content _ html|safe|truncatewords _
html:50 }}
•         </section>
•         <footer>
•           <a href="{% url 'articles-detail' article.slug
%}">
•             Read more
•           </a>
•         </footer>
•       {% endif %}
•       </article>
•     {% endfor %}
•     </section>
•   {% endblock %}
```

Not much here should be that surprising, but do take a look at how we are displaying the articles. We are using the 'forloop.counter' feature to display the first two articles differently than the the other eight. While the counter is less than or equal to two (for the first two articles) we will display an additional section that contains an article 'summary' and a 'Read more' link.

Notice also that we are marking the article contents safe so they render as HTML and using the 'truncatewords_html' filter to trim the article down to a 50-word summary. The 'truncatewords_html' filter is similar to the 'truncatewords' filter but works with HTML code instead of just plain text.

A better option though would be to



Top two categories

- [illegible]
- [illegible]

Our latest articles

Magnam cumque qui nobis voluptatem ex

By [illegible]

[illegible]

Read more

Dolorem et harum at enim

By [illegible]

[illegible]

Read more

Voluptatem delectus ut qui commodi eaque

By [illegible]

Voluptatibus repudiandae aliquam sed qui facilis

By [illegible]

Fugiat recusandae nesciunt est et in omnis adipisci

By [illegible]

Fugiat tenetur minima quis labore rerum porro

By [illegible]

Voluptatem asperiores neque et

By [illegible]

Nesciunt libero dolor laboriosam quaerat necessitatibus quas qui

By [illegible]

Sint quas dolore porro hic ut

By [illegible]

Enim saepe sunt aut rerum iste

By [illegible]

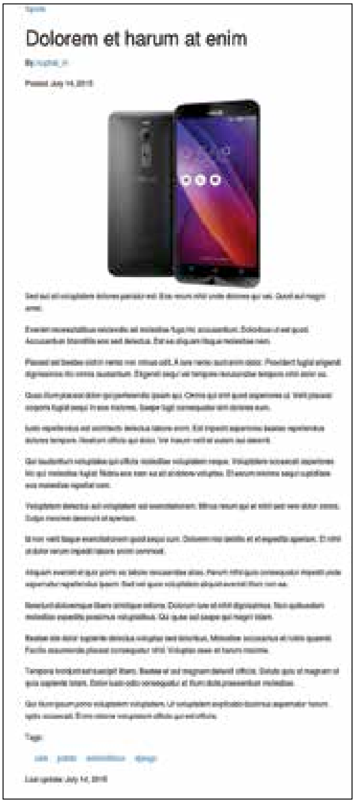
Our home page

have a separate field for the summary, since this usually wont lead to a useful summary.

cms/article_detail.html

```
{% extends 'site_base.html' %}
{% block title %}
  {{ article.title }}
{% endblock %}
{% block content %}
  <article>
  <header>
  <p>
 <a href="{% url 'categories-detail' article.category.slug
%}">
      {{ article.category }}
    </a>
  </p>
  <h1>{{ article.title }}</h1>
  <p>
  By:
  <a href="{% url 'authors-detail' article.author.username
%}">
    {{ article.author }}
  </a>
  </p>
  <p>
    Posted: {{ article.creation_date|date }}
  </p>
  <img src="{{ article.featured_image.url }}"
       alt="{{ article.title }}"/>
  </header>
  <section>
    {{ article.content_html|safe }}
  </section>
  <footer>
  <p>Tags:
    {% include '_tags.html' with tags=article.tags.all %}
  </p>
  <p>
    Last update: {{ article.edit_date|date }}
  </p>
  </footer>
  </article>
{% endblock %}
```

While this is certainly a more complex view, there isn’t much remarkable about it. Once again we are marking our article’s HTML content safe since it is generated by trusted people. You might also want to note how we are displaying the article’s featured image and how we are rendering this articles tags by including an external template. Let’s take a look at that template next.



Dolorem et harum at enim

Tags:

[illegible]

The article content page

_tags.html

```
<ul class=”inline-list”>
{% for tag in tags %}
  <li>
    <a href=”{% url ‘tags-detail’ tag.slug %}”>
      {{ tag.name }}
    </a>
  </li>
{% endfor %}
</ul>
```

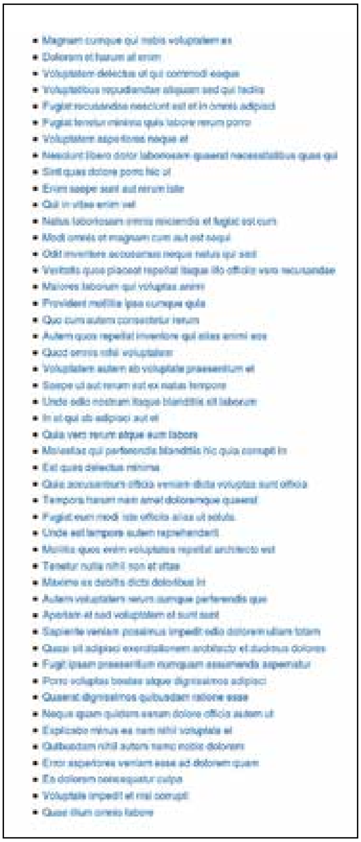
This small template snippet is meant to be included anywhere that a list of tags is required. In fact by changing some names around we can make this generic enough that it could work to generate any list of any kind of item.

cms/article_list.html

```
{% extends ‘site_base.html’ %}
{% block title %}
  Article List
{% endblock %}
{% block content %}
  <ul>
  {% for article in object_list %}
```

```
    <li>
      <a href="{% url 'articles-detail' article.slug %}">
        {{ article.title }}
      </a>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```

This is the first of the templates that is showing a list of items, so its important to note how that list of items is being provided to the template by the generic ListView. As you can see above it is passing along the list to the template as `object_list`. It is quite easily possible to change this to something more appropriate, such as `article_list` by configuring our ArticleList view. You can learn how from the Django documentation.



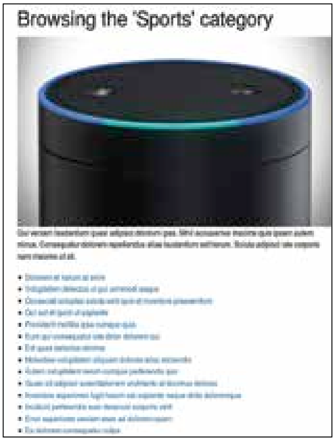
- Magnam cumque qui nobis voluptatem ex
- Dolorem et harum at enim
- Voluptatem delectus ut qui commodi eaque
- Voluptatibus repudiandae aliquam sed qui facilis
- Fugiat recusandae nesciunt est et in omnis adipisci
- Fugiat tenetur minima quis labore rerum porro
- Voluptatem asperiores neque et
- Nesciunt libero dolor laboriosam quaerat necessitatibus quas qui
- Sint quas dolore porro hic ut
- Enim saepe sunt aut rerum iste
- Qui in vitae enim vel
- Natus laboriosam omnis reiciendis et fugiat est cum
- Modi omnis et magnam cum aut est sequi
- Odit inventore accusamus neque natus qui sed
- Veritatis quos placeat repellat itaque illo officiis vero recusandae
- Maiores laborum qui voluptas animi
- Provident mollitia ipsa cumque quia
- Quo cum autem consectetur rerum
- Autem quos repellat inventore qui alias animi eos
- Quod omnis nihil voluptatem
- Voluptatem autem ab voluptate praesentium et
- Saepe ut aut rerum est ex natus tempore
- Unde odio nostrum itaque blanditiis sit laborum
- In ut qui ab adipisci aut et
- Quia vero rerum atque eum labore
- Molestias qui perferendis blanditiis hic quia corrupti in
- Est quis delectus minima
- Quia accusantium officia veniam dicta voluptas sunt officia
- Tempora harum nam amet doloremque quaerat
- Fugiat eum modi iste officiis alias ut soluta
- Unde est tempore autem reprehenderit
- Mollitia quos enim voluptates repellat architecto est
- Tenetur nulla nihil non et vitae
- Maxime ex debitis dicta doloribus in
- Autem voluptatem rerum cumque perferendis quo
- Aperiam et sed voluptatem et sunt sunt
- Sapiente veniam possimus impedit odio dolorem ullam totam
- Quasi sit adipisci exercitationem architecto et ducimus dolores
- Fugit ipsam praesentium numquam assumenda aspernatur
- Porro voluptas beatae atque dignissimos adipisci
- Quaerat dignissimos quibusdam ratione esse
- Neque quam quidem earum dolore officia autem ut
- Explicabo minus ea nam nihil voluptate et
- Quibusdam nihil autem nemo nobis dolorem
- Error asperiores veniam esse ad dolorem quam
- Ea dolorem consequatur culpa
- Voluptate impedit et nisi corrupti
- Quasi illum omnis labore

Our article listing page

cms/category_detail.html

```
{% extends 'site_base.html' %}
{% block title %}
  Browsing: {{ category.name }}
{% endblock %}
{% block content %}
  <header>
    <h1>Browsing the '{{ category.name }}'
      category</h1>
  </header>
  <section>
  <img src="{{ category.image.url }}"
       alt="{{ category.name }}"/>
  <p>{{ category.description }}</p>
  </section>
  <section>
  <ul>
  {% for article in category.article _ set.all %}
    <li>
      <a href="{% url 'articles-detail' article.slug %}">
        {{ article.title }}
      </a>
    </li>
  {% endfor %}
  </ul>
  </section>
{% endblock %}
```

One interesting thing to note in this code is the 'for' loop, or more precisely the list it is operating on. Our category model says nothing about its relationship to articles, however our article does point to categories. Django uses this relationship to automatically link from categories to articles by providing a convenient way to get to a list of all the articles that point to that category. This set of articles is accessible via the `article_set` property of a category.

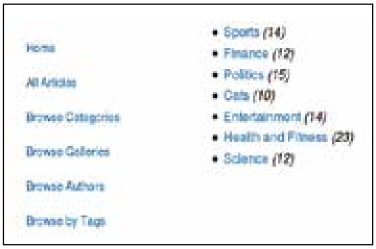


Browsing the 'Sports' category

Our category browsing page

cms/category_list.html

```
{% extends 'site _ base.html' %}
{% block title %}
  Categories
{% endblock %}
{% block content %}
  <ul>
  {% for category in object _ list %}
    <li>
    <a href="{% url 'categories-detail' category.slug %}">
      {{ category.name }}
    </a>
    <em>({{ category.article _ set.count }})</em>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```



Our category listing page

Just to make things interesting — now that you know about the 'article_set' property — we are decided to add an extra bit of information about categories in the category listing page. You will see above that we are printing the value of 'category.article_set.count'. The count method returns the number of objects in any list, so this value is printing the number of articles that belong to a category.

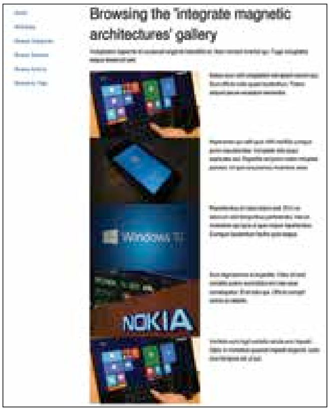
Note that a better way to accomplish this is to use annotations; this is highly inefficient!

cms/gallery_detail_base.html

```
{% extends ‘site_base.html’ %}
{% block title %}
  {{ gallery.name }}
{% endblock %}
{% block content %}
  <header>
    <h1>Browsing the ‘{{ gallery.name }}’
      gallery</h1>
  </header>
  <section>
    <p>{{ gallery.description }}</p>
  </section>
  <section>
  {% block gallery %}
    <ul class=”small-block-grid-3”>
    {% for image in gallery.images.all %}
      <li>
      <a href=”{% url ‘image-detail’ image.slug %}”>
        <img src=”{{ image.image.url }}”
             alt=”{{ image.name }}”/>
      </a>
      </li>
    {% endfor %}
    </ul>
  {% endblock %}
  </section>
{% endblock %}
```

This is our base template for the gallery's three layouts. Notice how we have added another block within our content block in this template. When we override this template for the three layouts we will only need to fill the contents in this block.

The code we have placed in this block above is for one of the layouts (the linked layout) so that in that particular template we don't need to do anything other than to extend from this one.


Browsing the 'integrate magnetic architectures' gallery

The horizontal layout of our gallery page

cms/gallery_detail_horizontal.html

```
{% extends 'cms/gallery_detail_base.html' %}
{% block gallery %}
  <ul class="no-bullet">
  {% for image in gallery.images.all %}
    <li>
    <figure class="row">
    <img src="{{ image.image.url }}"
         alt="{{ image.alt_text }}"
         class="small-6 columns"/>
    <figcaption class="small-6 columns">
      {{ image.alt_text }}
    </figcaption>
    </figure>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```

cms/gallery_detail_vertical.html

```
{% extends 'cms/gallery_detail_base.html' %}
{% block gallery %}
  <ul class="no-bullet">
  {% for image in gallery.images.all %}
    <li>
```

```
      <figure>
      <img class="small-12 columns"
           src="{{ image.image.url
}}"
           alt="{{ image.alt _ text
}}"/>
      <figcaption class="small-12
columns">
        {{ image.alt _ text }}
      </figcaption>
      </figure>
      </li>
    {% endfor %}
    </ul>
  {% endblock %}
```

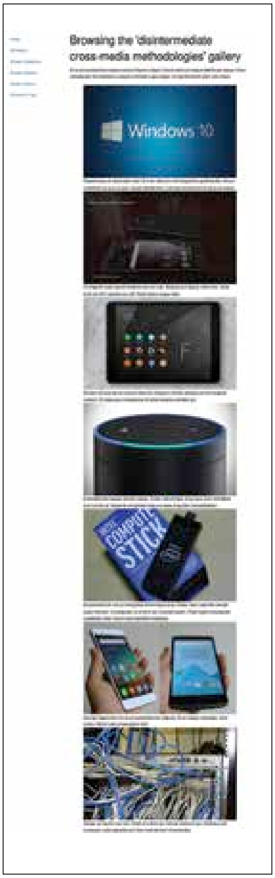
cms/gallery_detail_links.html

```
{% extends 'cms/gallery _ detail _
base.html' %}
```

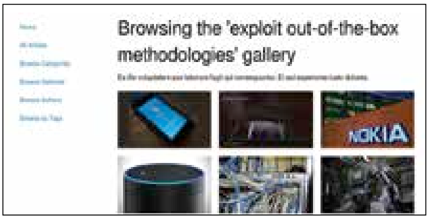
Since we already included this template in the gallery block in the base template, we don't need to write any code here! If only all templates were like this.

Technically these three gallery layouts could have been accomplished by just using different CSS code, but this helped us explore more Django features rather than getting into the nitty-gritty of CSS. Oh well, the things we do to teach.

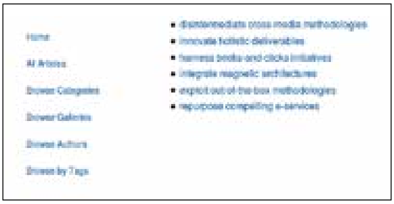
We could have also placed this gallery code into separate HTML files (like we have now) and just included the correct one by checking the value of `gallery.layout` in the template code. Or even had the entire code in a single template file in between `if`, `elseif`, and `else` clauses. This method just seems a lot cleaner.


Browsing the 'disintermediate cross-media methodologies' gallery

The vertical layout of our gallery page


Browsing the 'exploit out-of-the-box methodologies' gallery

The linked thumbnails layout of our gallery page



Home
All Articles
Browse Categories
Browse Galleries
Browse Authors
Browse by Tags

- disintermediate cross media methodologies
- innovate holistic deliverables
- harness bricks-and-clicks initiatives
- integrate magnetic architectures
- exploit out-of-the-box methodologies
- repurpose compelling e-services

Our gallery listing page

cms/gallery_list.html

```
{% extends ‘site_base.
html’ %}
{% block title %}
  Galleries
{% endblock %}
{% block content %}
  <ul>
  {% for gallery in object_list %}
    <li>
      <a href=”{% url ‘galleries-detail’ gallery.slug %}”>
        {{ gallery.name }}
      </a>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```

This is probably one of the least spectacular templates we could have made for a list of galleries. A list of links that point to galleries? Seriously!?

We wouldn’t be very good teachers though if we spoon-fed you everything now would we! There is a lot of scope for improvement in this view, for instance you could have each gallery display its first image here, or a the first three thumbnails, or anything better than this really.

Our image display page. You can click on any image in the gallery layout with linked thumbnails to see this

cms/image_detail.html

```
{% extends ‘site_base.html’ %}
{% block title %}
  Image: {{ image.name }}
{% endblock %}
{% block content %}
  <figure>
    <img src=”{{ image.image.url }}”
         alt=”{{ image.alt_text }}”/>
    <figcaption>{{ image.alt _ text }}</figcaption>
  </figure>
{% endblock %}
```

cms/tag_detail.html

```
{% extends 'site _ base.html' %}
{% block title %}
  Browsing: {{ tag.name }}
{% endblock %}
{% block content %}
  <header>
    <h1>Browsing article with the '{{ tag.name }}'
      tag</h1>
  </header>
  <section>
  <ul>
  {% for article in tag.article _ set.all %}
    <li>
      <a href="{% url 'articles-detail' article.slug %}">
        {{ article.title }}
      </a>
    </li>
  {% endfor %}
  </ul>
  </section>
{% endblock %}
```

Here we once again see `article_set` in use, and we can see a pattern. Like the Category model, the Tag model does not specify a link to an Article, rather an Article specifies a link to the Tag model. Django automatically created this reverse link.

cats twitter selfie potato apple
ludonarrative finance facebook google android linux python
django musical rabbit cyborgs cms
Our tag browsing page

cats twitter selfie potato apple
ludonarrative finance facebook google android linux python
django musical rabbit cyborgs cms
Our tag cloud page

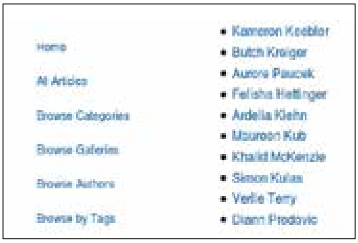
cms/tag_list.html

```
{% extends 'site _ base.html' %}
{% block title %}
```

```
  Tag Cloud
{% endblock %}
{% block content %}
  <ul>
  {% for tag in object _ list %}
    <li style="font-size: {{ tag.num _ articles }}pt;
               display: inline-block;">
      <a href="{% url 'tags-detail' tag.slug %}">
        {{ tag.name }}
      </a>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```

Even if you forgive our use of inline style — which you should always strive to avoid — this is a rather naive implementation of a tag cloud. In this above implementation the font size of a tag is equal to the number of articles that have that tag. You can probably imagine what will happen when there are thousands of articles, with tags that have been applied hundreds of times.

A proper tag cloud system would clip the range of font sizes to something more reasonable, so the largest tag sizes aren't hundreds of points and the smallest ones are unreadably small.



- Kameron Koebler
- Butch Kreiger
- Aurora Paucek
- Felisha Hettinger
- Ardella Klehn
- Maureen Kub
- Khalid McKenzie
- Simon Kulas
- Verlie Terry
- Diann Prodovic

Our author profile page

auth/user_detail.html

```
{% extends 'site _ base.
html' %}
{% block title %}
  Author: {{ user.get _ full _ name }}
{% endblock %}
{% block content %}
  <header>
    <h1>Articles by {{ user.get _ full _ name }}</h1>
  </header>
  <section>
  <ul>
  {% for article in user.article _ set.all %}
    <li>
```

```
        <a href=”{% url ‘articles-detail’ article.slug %}”>
          {{ article.title }}
        </a>
      </li>
    {% endfor %}
    </ul>
    </section>
{% endblock %}
```

The most important thing to notice about this template is that it isn’t supposed to be placed in the ‘cms’ subdirectory of the ‘templates’ folder since the user model isn’t part of our CMS app, but instead comes with the `auth` application that ships with Django, hence its placement in the ‘auth’ folder.

Also note that this file is called ‘user_detail.html’ and not ‘author_detail.html’ since we are dealing with the user model. They might be authors to us, but for Django they are just CMS users.

Our author browsing page

auth/user_list.html

```
{% extends ‘site _ base.html’ %}
{% block title %}
  Authors
{% endblock %}
{% block content %}
  <ul>
  {% for author in object _ list %}
    <li>
      <a href=”{% url ‘authors-detail’ author.username 
%}”>
        {{ author.get _ full _ name }}
      </a>
    </li>
  {% endfor %}
  </ul>
{% endblock %}
```

For the same reason as with ‘user_detail.html’ template, this template too needs to be placed in the ‘auth’ folder.

DJANGO ADMIN

Django's admin framework is a marvellous piece of software that can save you tonnes of time in building the bits of your site that no user will see anyway: the admin panels.

We now near the end dear reader, so let's make the best of them. Let's now code an entire administrative interface to list, search, filter, add, update, and delete the data in our CMS. Till now there wasn't a way to add new content to our website using a browser, which would make this a poor CMS indeed. We've been having so much fun we didn't even notice.

Django includes an applications just for automatically building an administrative interface for any kind of website with any kind of data. The least it needs from you is to just tell it to show an interface for your models, but you can customise the admin panel to a large extent to make it easier for your users to mange your content — which is the actual purpose of a CMS after all.

Before we build our admin panel, let's check it out in its current state first.

Accessing the Admin Panel

Back when we were discussing the URL structure and configuration of our website, we glossed over this odd entry that seemed to be there by default:

```
url(r'^admin/', include(admin.site.urls)),
```

We mentioned that this pattern was intercepting any URLs starting with 'admin/' and passing them along to the admin app that ships with Django.

As a side note, we can do something similar with our our CMS app, which is to move all our CMS URLs to a 'urls.py' file in our 'cms' app folder and include it as above.

Coming back to our admin panel, if you try to open it now by visiting '*http://localhost:8000/admin/*' you will be faced with a login dialog and no way to register a new account. The admin panel allows us to create new users and log in, but to use the admin panel we first need to be able to log in. A catch-22, one was about due by now.

So how do we log in? To break from this cycle Django includes a management command called 'createsuperuser' that walks you through creating a new admin account. It can be use as follows:

```
python3 manage.py createsuperuser
```

When you execute the above command it will ask you for a username, an email and a password; these are basic requirements of any user account. With this new user created, you can now log in using the admin URL.

In its current state you will see that the admin panel will have two models to edit, Users and Groups. Try them both, try creating new users and groups, edit your newly created user account.

Any users you create will not be able to access this admin panel by default, even if you give them the permission to do so. To give users access to the admin panel you need to mark them as a staff. Only staff users can log into the admin panel. Being able to log into the admin panel doesn't give them the ability to do anything else though. Unless given correct permissions they will not be able to add / edit / remove anything.

A quick way to give a user permission to do everything is to mark them as a superuser. This gives the user all permissions automatically.

Using our Models via the Admin Site

What we'd really like to see on the admin panel though are our own models, so we can add articles, images, galleries and tags. It turns out this is very simple, just register the models with the admin site framework in the 'admin.py' file. You can register a model with the admin framework as follows:

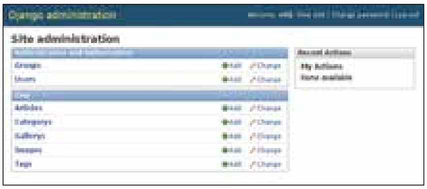
```
admin.site.register(Article)
```

We can register all our models in just a few lines of code:

```
from django.contrib import admin
from cms.models import *
```

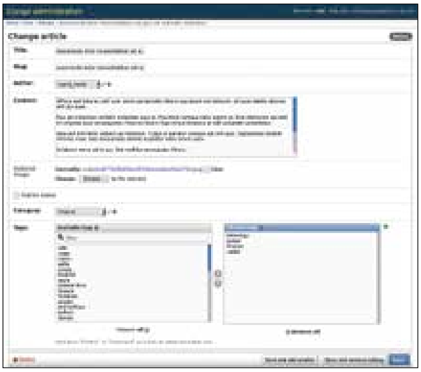
- `admin.site.register(Article)`
- `admin.site.register(Tag)`
- `admin.site.register(Category)`
- `admin.site.register(Gallery)`
- `admin.site.register(Image)`

Now visit the admin website for a pleasant surprise; everything just works. You can add articles, see a list of articles, click on one in the list and you can edit it. When you try to add a new article, you will see a from where you can enter a title, a slug, and the text content of the article. You will be able to see a list of authors and select one from the list; clicking on the small plus next to the author drop-down list to show a pop-up window that allows you to add a new user to select as an author. You'll probably want reserve the permission to add users to admins, or at least restrict it highly, and once you do so, this plus sign will disappear!



The Django admin panel

For the featured image, you will see a button that shows a browse dialog when clicked so you can select a file to upload. The publish status is a simple checkbox and selecting or adding a category is a process similar to selecting / adding an author.

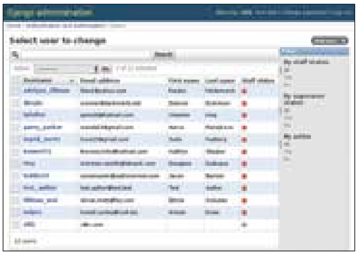


Editing an existing article in our CMS

The tags interface isn't exactly the most convenient. If there are too many tags as you have a scroll through hundreds of tags and keep Ctrl pressed while selecting the tags you want to include. Also manually having to enter a slug is inconvenient, it would make most sense for the slug to be auto-generated.

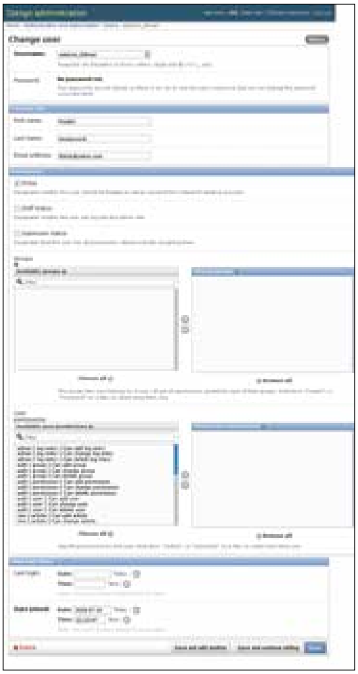
Still, think of the fact that all this is automatically generated using our the fields in our model and nothing else! Character fields have a text input field for them, the text field gets a larger text edit area, boolean fields have a checkbox, foreign keys show a dropdown, and ManyToMany fields show a multi-select list box. If we had a date field we'd even see a date entry and selection widget.

We can customise a lot about this page, how the different fields are arranged, how author, category and tag selection works, and even have the slug be autogenerated. To see a customised page, look at the page that lets you edit users. It has fields sectioned into categories such as 'Personal info', 'Permissions' and 'Important dates'. Each field on this page has some help text associated with it. Even the widgets that let you select permissions is a much better system for something like tags than what we have on our page.



The Django user listing page that ships with Django's auth framework

Now if come to the page listing articles, and compare it to the page listing users. See how there is a search box to search for users, but there isn't one for articles. Also see how the user page has all these columns that have useful information, and the article page just has the article name. Also see how you can filter users based on whether they have staff status, superuser status and active status. Again, all missing in the article page.



The Django user add /edit page

We keep using the article section as an example but these features are missing for all our models. The good news is that we can add all the fancy features that you have with the user and group models for any of our own models, it's just a matter of configuring them, there isn't even much code required!

Customising our Admin Pages

To enable all these great features in our own admin pages we need to create an class that configures these features include it as the second parameter while using `admin.site.register()` to register our model with the admin site. This class needs to be a subclass of Django's AdminModel class. We'll call this class `ArticleAdmin`.

Now if we want to display the author, category, creation date, update date and publish status of our articles in columns we can simply add an attribute called `list_display` in our class definition of ArticleAdmin that is a tuple containing all the columns to show:

```
class ArticleAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'category',
                    'creation_date', 'edit_date',
                    'publish_status',)

admin.site.register(Article, ArticleAdmin)
```

This is all it takes for this feature! Now let's add a couple of more features to the list page and then see the code that results. After that we can see about customising that edit page.

Firstly, why should you have to open the article edit page to simply switch it from published to unpublished or vice versa? To enable editing a field from the list page, add another attribute to this class called `list_editable` that has a tuple with all editable fields. Do that and you will see the ability to change those fields from this page, and then save the changes by clicking on the save button at the bottom of the page.

For the filtering function we saw in the user list page, just add an attribute called 'list_filter' and set it to a tuple listing all the fields you want to filter on.

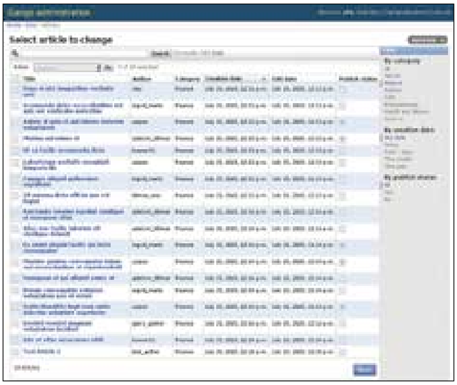
For adding search functionality, add a `search_fields` attribute with a list of all the fields to search in. There is scope for additional configuration here that you can look up in the Django documentation.

Here is what our ArticleAdmin class looks like right now:

```
class ArticleAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'category',
                    'creation_date', 'edit_date',
                    'publish_status',)
    list_editable = ('publish_status',)
    list_filter = ('category',
                   'creation_date',
                          'publish_status',)
      search_fields = ['author__first_name',
                          'author__last_name',
                          'author__email',
                          'author__username',
                          'title', 'content', ]
```

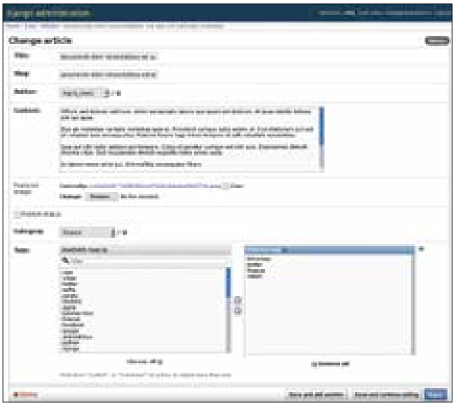
In the above code it is very important that there be a comma after 'publish_status' since it is the sole value in that tuple and the presence of that comma differentiates between it being just a string value and a tuple with a single string value.

Moving on to the article edit page. Let's fist get rid of that horrible multiple select system for our tags and switch to the system used by user edit page for permissions and groups. This is a simple matter of setting `filter_horizontal` to list that as `tags` as the sole entry (it is after all the only ManytoManyField present in this model). Alternatively you can use `filter_vertical`, they just use different templates for the same job.



Our new article listing page enhanced with searching filtering and quick editing functionality

To have that slug field automatically filled based on the title field, we can add an attribute called `prepopulated_fields` with the value '{'slug': ('title',)}'. This value looks a little strange, but it is essentially a dictionary containing the names of the fields to be autofilled



Our new article edit / add page

(in this case only 'slug') set to a tuple of all the source fields to use to autofill it (in this case only 'title').

```
class ArticleAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'category',
                    'creation_date', 'edit_date',
                    'publish_status',)
    list_editable = ('publish_status',)
    list_filter = ('category',
                   'creation_date',
                   'publish_status',)
    search_fields = ['author__first_name',
                     'author__last_name',
                     'author__email',
                     'author__username',
                     'title', 'content', ]

    filter_horizontal = ['tags']
    prepopulated_fields = {'slug': ('title',)}
```

There is a lot more that you can still do to make the experience better. For one you can group filed into sections like in the user edit page and add help text. You can even add additional classes and CSS / JavaScript code to improve the functionality of the admin page. For instance you could include a markdown editor for the content field to make article composition easier.

That all however shall have to be left as an exercise to you dear reader.

CONCLUSION

In our day we had to write a CMS in assembly using a keyboard with just a 0 and 1 keys. Now we've spoiled you

One of the goals of this text, one that we have left unmentioned until now, is to avoid the use of any third party packages and applications that are designed to extend the functionality of Django or to provide entire apps. Those are certainly useful and you should probably use them whenever you can instead of writing stuff from scratch.

However it was important to the author that this text be as raw a depiction of Django as possible, so at no point should you be confused if something is a part of Django or not. It was also to show the raw power of Django when it comes to developing such apps.

As such there are only three Python packages that we used in this entire book. The first is Django itself, and the second is Pillow, an image manipulation library for Python that Django requires for some of its features. The final external library we have used is Markdown. This is not something supported out-of-the-box by either Python or Django, but an integration like this one highlights that even if something is missing from Django, it isn't hard to add.

It's quite obvious that what we have designed is no killer CMS that will take over the world. There are tonnes of places where it has scope for betterment. There are inefficiencies such as repeatedly generating HTML from markdown instead of generating it each time article is saved and saving it to the database. Or using complex database queries in templates that will cause hundreds of queries to the database each time a page is loaded. Our templates are quite raw and unpleasant, and don't even take advantage of the little data we do have. Pages other than the main and articles list page will have links to articles that are unpublished, and as such can't be opened.

We also briefly mentioned browsing articles by date, but this is something we never implemented, why don't you start there? (Hint: look into overriding the 'get_queryset' method)

Yet each one of these failing is a chance for you to hone your newfound skills. Now that this text comes to a conclusion, we hope that you are left with a good understanding of how Django applications are structured and how to get things working the way you want.

digit.in/review

NEWS AND REVIEWS

Comprehensive news, unbiased re-views and ratings to help you make the right purchase

digit.in/download

DOWNLOADS

Check out the latest software downloads, games for your Win-dows, Linux, Mac and PDA/mobiles

All this and more in the **world of Technology**

VISIT NOW digit.in

RNI REGN. No. MAH ENG/2001/3791

www.facebook.com

Join 500K+ members of the digit community

http://www.facebook.com/thinkdigit

facebook

digit.in

Digit

Your favourite magazine on your social network. Interact with thousands of fellow Digit readers.

http://www.facebook.com/IThinkGadgets

facebook

I Think Gadgets

An active community for those of you who love mobiles, laptops, cameras and other gadgets. Learn and share more on technology.

http://www.facebook.com/GladtobeaProgrammer

facebook

Glad to be a Programmer

If you enjoy writing code, this community is for you. Be a part and find your way through development.

http://www.facebook.com/devworx.in

facebook

Devworx

devworx, a niche community for software developers in India, is supported by 9.9 Media, publishers of Digit