

Django Web Framework (Python)

Page url: <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django>

Django is an extremely popular and fully featured server-side web framework, written in Python. The module shows you why Django is one of the most popular web server frameworks, how to set up a development environment, and how to get started with using it to create your own web applications.

[Prerequisites](#)[Edit](#)

Before starting this module you don't need to have any knowledge of Django. You will need to understand what server-side web programming and web frameworks are, ideally by reading the topics in our [Server-side website programming first steps](#) module.

A general knowledge of programming concepts and [Python](#) is recommended, but not essential to understanding the core concepts.

Note: Python is one of the easiest programming languages for novices to read and understand. That said, if you want to understand this module better then there are numerous free books and tutorials available on the Internet (new programmers might want to check out the [Python for Non Programmers](#) page on the python.org wiki).

[Guides](#)[Edit](#)

[Django introduction](#)

In this first Django article we answer the question "What is Django?" and give you an overview of what makes this web framework special. We'll outline the main features, including some of the advanced functionality that we won't have time to cover in detail in this module. We'll also show you some of the main building blocks of a Django application, to give you an idea of what it can do before you then go on to set it up and start playing.

[Setting up a Django development environment](#)

Now that you know what Django is for, we'll show you how to setup and test a Django development environment on Windows, Linux (Ubuntu), and Mac OS X — whatever common operating system you are using, this article should give you what you need to be able to start developing Django apps.

[Django Tutorial: The Local Library website](#)

The first article in our practical tutorial series explains what you'll learn, and provides an overview of the "local library" example website we'll be working through and evolving in subsequent articles.

[Django Tutorial Part 2: Creating a skeleton website](#)

This article shows how you can create a "skeleton" website project as a basis, which you can then go on to populate with site-specific settings, urls, models, views, and templates.

[Django Tutorial Part 3: Using models](#)

This article shows how to define models for the *LocalLibrary* website — models represent the data structures we want to store our app's data in, and also allow Django to store data in a database for us (and modify it later on). It explains what a model is, how it is declared, and some of the main field types. It also briefly shows a few of the main ways you can access model data.

[Django Tutorial Part 4: Django admin site](#)

Now that we've created models for the *LocalLibrary* website, we'll use the Django Admin site to add some "real" book data. First we'll show you how to register the models with the admin site, then we'll show you how to login and create some data. At the end we show some of ways you can further improve the presentation of the admin site.

[Django Tutorial Part 5: Creating our home page](#)

We're now ready to add the code to display our first full page — a home page for the *LocalLibrary* that shows how many records we have of each model type and provides sidebar navigation links to our other pages. Along the way we'll gain practical experience in writing basic URL maps and views, getting records from the database, and using templates.

[Django Tutorial Part 6: Generic list and detail views](#)

This tutorial extends our *LocalLibrary* website, adding list and detail pages for books and authors. Here we'll learn about generic class-based views, and show how they can reduce the amount of code you have to write for common use cases. We'll also go into URL handling in greater detail, showing how to perform basic pattern matching.

[Django Tutorial Part 7: Sessions framework](#)

This tutorial extends our *LocalLibrary* website, adding a session-based visit-counter to the home page. This is a relatively simple example, but it does show how you can use the session framework to provide persistent behaviour for anonymous users in your own sites.

[Django Tutorial Part 8: User authentication and permissions](#)

In this tutorial we'll show you how to allow users to login to your site with their own accounts, and how to control what they can do and see based on whether or not they are logged in and their *permissions*. As part of this demonstration we'll extend the *LocalLibrary* website, adding login and logout pages, and user- and staff-specific pages for viewing books that have been borrowed.

[Django Tutorial Part 9: Working with forms](#)

In this tutorial we'll show you how to work with [HTML Forms](#) in Django, and in particular the easiest way to write forms to create, update, and delete model instances. As part of this demonstration we'll extend the *LocalLibrary* website so that librarians can renew books, and create, update, and delete authors using our own forms (rather than using the admin application).

[Django Tutorial Part 10: Testing a Django web application](#)

As websites grow they become harder to test manually — not only is there more to test, but, as the interactions between components become more complex, a small change in one area can require many additional tests to verify its impact on other areas. One way to mitigate these problems is to write automated tests, which can easily and reliably be run every time you make a change. This tutorial shows how to automate *unit testing* of your website using Django's test framework.

[Django Tutorial Part 11: Deploying Django to production](#)

Now you've created (and tested) an awesome *LocalLibrary* website, you're going to want to install it on a public web server so that it can be accessed by library staff and members over the Internet. This article provides an overview of how you might go about finding a host to deploy your website, and what you need to do in order to get your site ready for production.

[Django web application security](#)

Protecting user data is an essential part of any website design. We previously explained some of the more common security threats in the article [Web security](#) — this article provides a practical demonstration of how Django's in-built protections handle such threats.

Django introduction

In this first Django article we answer the question "What is Django?" and give you an overview of what makes this web framework special. We'll outline the main features, including some of the advanced functionality that we won't have time to cover in detail in this module. We'll also show you some of the main building blocks of a Django application (although at this point you won't yet have a development environment in which to test it).

Prerequisites: Basic computer literacy. A general understanding of [server-side website programming](#), and in particular the mechanics of [client-server interactions in websites](#).

Objective: To gain familiarity with what Django is, what functionality it provides, and the main building blocks of a Django application.

[What is Django?](#)[Edit](#)

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It is free and open source, has a thriving and active community, great documentation, and many options for free and paid-for support.

Django helps you write software that is:

Complete

Django follows the "Batteries included" philosophy and provides almost everything developers might want to do "out of the box". Because everything you need is part of the one "product", it all works seamlessly together, follows consistent design principles, and has extensive and [up-to-date documentation](#).

Versatile

Django can be (and has been) used to build almost any type of website — from content management systems and wikis, through to social networks and news sites. It can work with any client-side framework, and can deliver content in almost any format (including HTML, RSS feeds, JSON, XML, etc). The site you are currently reading is based on Django!

Internally, while it provides choices for almost any functionality you might want (e.g. several popular databases, templating engines, etc.), it can also be extended to use other components if needed.

Secure

Django helps developers avoid many common security mistakes by providing a framework that has been engineered to "do the right things" to protect the website automatically. For example, Django provides a secure way to manage user accounts and passwords, avoiding common mistakes like putting session information in cookies where it is vulnerable (instead cookies just contain a key, and the actual data is stored in the database) or directly storing passwords rather than a password hash.

A password hash is a fixed-length value created by sending the password through a [cryptographic hash function](#). Django can check if an entered password is correct by running it through the hash function and comparing the output to the stored hash value. However due to the "one-way" nature of the function, even if a stored hash value is compromised it is hard for an attacker to work out the original password.

Django enables protection against many vulnerabilities by default, including SQL injection, cross-site scripting, cross-site request forgery and clickjacking (see [Website security](#) for more details of such attacks).

Scalable

Django uses a component-based "[shared-nothing](#)" architecture (each part of the architecture is independent of the others, and can hence be replaced or changed if needed). Having a clear separation between the different parts means that it can scale for increased traffic by adding hardware at any level: caching servers, database servers, or application servers. Some of the busiest sites have successfully scaled Django to meet their demands (e.g. Instagram and Disqus, to name just two).

Maintainable

Django code is written using design principles and patterns that encourage the creation of maintainable and reusable code. In particular, it makes use of the Don't Repeat Yourself (DRY) principle so there is no unnecessary duplication, reducing the amount of code. Django also promotes the grouping of related functionality into reusable "applications" and, at a lower level, groups related code into modules (along the lines of the [Model View Controller \(MVC\)](#) pattern).

Portable

Django is written in Python, which runs on many platforms. That means that you are not tied to any particular server platform, and can run your applications on many flavours of Linux, Windows, and Mac OS X. Furthermore, Django is well-supported by many web hosting providers, who often provide specific infrastructure and documentation for hosting Django sites.

[Where did it come from?](#) [Edit](#)

Django was initially developed between 2003 and 2005 by a web team who were responsible for creating and maintaining newspaper websites. After creating a number of sites, the team began to factor out and reuse lot of common code and design patterns. This common code evolved into a generic web development framework, which was open-sourced as the "Django" project in July 2005.

Django has continued to grow and improve, from its first milestone release (1.0) in September 2008 through to the recently-released version 1.11 (2017). Each release has added new

functionality and bug fixes, ranging from support for new types of databases, template engines, and caching, through to the addition of "generic" view functions and classes (which reduce the amount of code that developers have to write for a number of programming tasks).

Note: Check out the [release notes](#) on the Django website to see what has changed in recent versions, and how much work is going into making Django better.

Django is now a thriving, collaborative open source project, with many thousands of users and contributors. While it does still have some features that reflect its origin, Django has evolved into a versatile framework that is capable of developing any type of website.

How popular is Django?[Edit](#)

There isn't any readily-available and definitive measurement of popularity of server-side frameworks (although sites like [Hot Frameworks](#) attempt to assess popularity using mechanisms like counting the number of GitHub projects and StackOverflow questions for each platform). A better question is whether Django is "popular enough" to avoid the problems of unpopular platforms. Is it continuing to evolve? Can you get help if you need it? Is there an opportunity for you to get paid work if you learn Django?

Based on the number of high profile sites that use Django, the number of people contributing to the codebase, and the number of people providing both free and paid for support, then yes, Django is a popular framework!

High-profile sites that use Django include: Disqus, Instagram, Knight Foundation, MacArthur Foundation, Mozilla, National Geographic, Open Knowledge Foundation, Pinterest, and Open Stack (source: [Django home page](#)).

Is Django opinionated?[Edit](#)

Web frameworks often refer to themselves as "opinionated" or "unopinionated".

Opinionated frameworks are those with opinions about the "right way" to handle any particular task. They often support rapid development *in a particular domain* (solving problems of a particular type) because the right way to do anything is usually well-understood and well-documented. However they can be less flexible at solving problems outside their main domain, and tend to offer fewer choices for what components and approaches they can use.

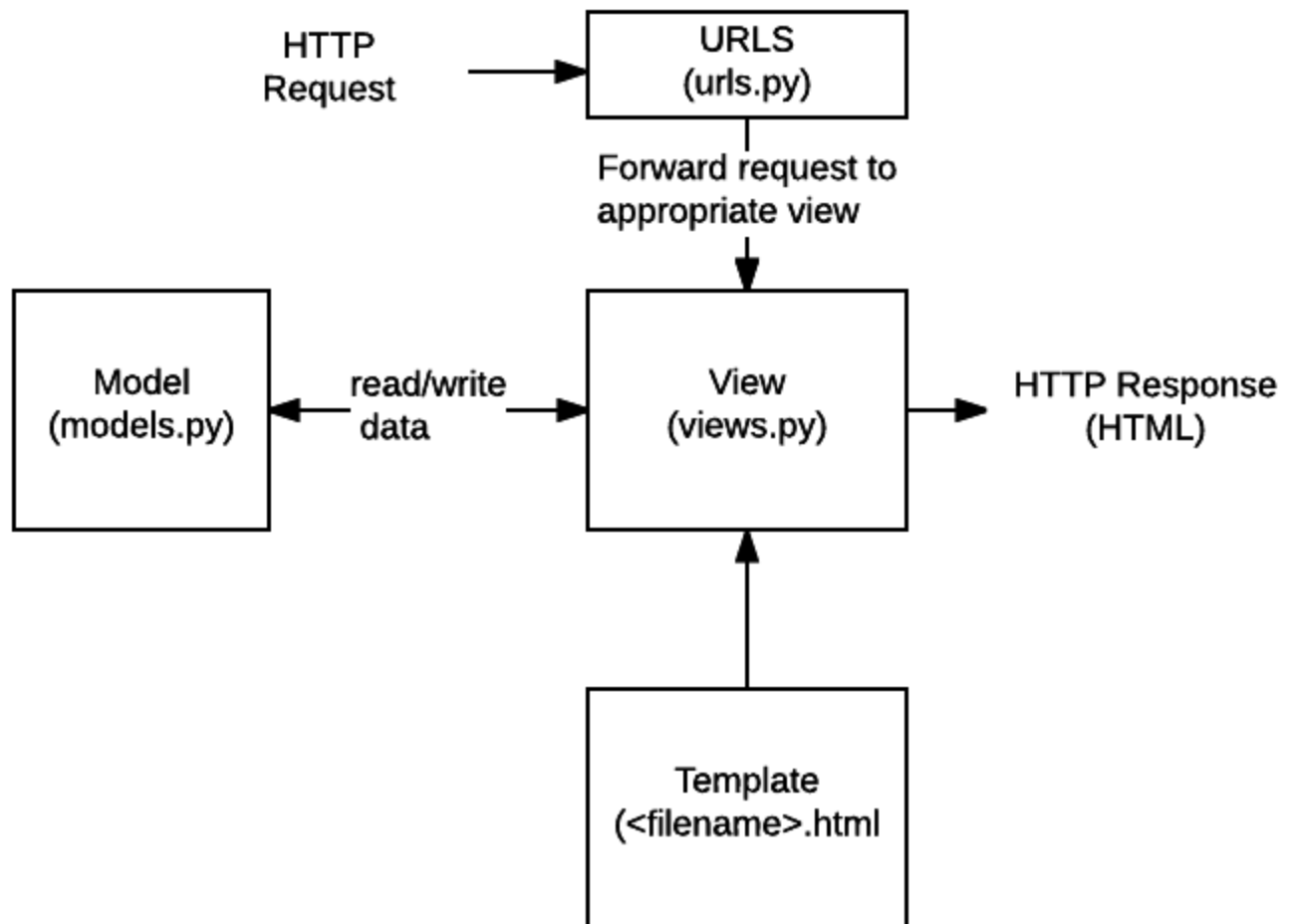
Unopinionated frameworks, by contrast, have far fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used. They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that you need to find those components yourself.

Django is "somewhat opinionated", and hence delivers the "best of both worlds". It provides a set of components to handle most web development tasks and one (or two) preferred ways to use them. However, Django's decoupled architecture means that you can usually pick and choose from a number of different options, or add support for completely new ones if desired.

What does Django code look like?[Edit](#)

In a traditional data-driven website, a web application waits for HTTP requests from the web browser (or other client). When a request is received the application works out what is needed based on the URL and possibly information in `POST` data or `GET` data. Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request. The application will then return a response to the web browser, often dynamically creating an HTML page for the browser to display by inserting the retrieved data into placeholders in an HTML template.

Django web applications typically group the code that handles each of these steps into separate files:



- **URLs:** While it is possible to process requests from every single URL via a single function, it is much more maintainable to write a separate view function to handle each resource. A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL. The URL mapper can also match particular patterns of strings or digits that appear in an URL, and pass these to a view function as data.
- **View:** A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via *models*, and delegate the formatting of the response to *templates*.

- **Models:** Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.
- **Templates:** A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A *view* can dynamically create an HTML page using an HTML template, populating it with data from a *model*. A template can be used to define the structure of any type of file; it doesn't have to be HTML!

Note: Django refers to this organisation as the "Model View Template (MVT)" architecture. It has many similarities to the more familiar [Model View Controller](#) architecture.

The sections below will give you an idea of what these main parts of a Django app look like (we'll go into more detail later on in the course, once we've set up a development environment).

[Sending the request to the right view \(urls.py\)](#)

A URL mapper is typically stored in a file named **urls.py**. In the example below, the mapper (`urlpatterns`) defines a list of mappings between specific URL *patterns* and corresponding view functions. If an HTTP Request is received that has a URL matching a specified pattern (e.g. `r'^$',` below) then the associated view function (e.g. `views.index`) will be called and passed the request.

```
urlpatterns = [
    url(r'^$', views.index),
    url(r'^([0-9]+)/$', views.best),
]
```

Note: A little bit of Python:

- The `urlpatterns` object is a list of `url()` functions. In Python, lists are defined using square brackets. Items are separated by commas and may have an [optional trailing comma](#). For example: `[item1, item2, item3,]`.
- The odd-looking syntax for the pattern is known as a regular expression. We'll talk about these in a later article!
- The second argument to `url()` is another function that will be called when the pattern is matched. The notation `views.index` indicates that the function is called `index()` and can be found in a module called `views` (i.e. inside a file named `views.py`).

[Handling the request \(views.py\)](#)

Views are the heart of the web application, receiving HTTP requests from web clients and returning HTTP responses. In between, they marshall the other resources of the framework to access databases, render templates, etc.

The example below shows a minimal view function `index()`, which could have been called by our URL mapper in the previous section. Like all view functions it receives an `HttpRequest` object as a parameter (`request`) and returns an `HttpResponse` object. In this case we don't do anything with the request, and our response simply returns a hard-coded string. We'll show you a request that does something more interesting in a later section.

```
## filename: views.py (Django view functions)
```



```

from django.http import HttpResponse

def index(request):
    # Get an HttpRequest - the request parameter
    # perform operations using information from the request.
    # Return HttpResponse
    return HttpResponse('Hello from Django!')

```

Note: A little bit of Python:

- [Python modules](#) are "libraries" of functions, stored in separate files, that we might want to use in our code. Here we import just the `HttpResponse` object from the `django.http` module so that we can use it in our view: `from django.http import HttpResponse`. There are other ways of importing some or all objects from a module.
- Functions are declared using the `def` keyword as shown above, with named parameters listed in brackets after the name of the function; the whole line ends in a colon. Note how the next lines are all **indented**. The indentation is important, as it specifies that the lines of code are inside that particular block (mandatory indentation is a key feature of Python, and is one reason that Python code is so easy to read).

Views are usually stored in a file called **views.py**.

Defining data models ([models.py](#))

Django web applications manage and query data through Python objects referred to as models. Models define the structure of stored data, including the field *types* and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.

The code snippet below shows a very simple Django model for a `Team` object. The `Team` class is derived from the `django class models.Model`. It defines the team name and team level as character fields and specifies a maximum number of characters to be stored for each record. The `team_level` can be one of several values, so we define it as a choice field and provide a mapping between choices to be displayed and data to be stored, along with a default value.

```

# filename: models.py

from django.db import models

class Team(models.Model):
    team_name = models.CharField(max_length=40)

    TEAM_LEVELS = (
        ('U09', 'Under 09s'),
        ('U10', 'Under 10s'),
        ('U11', 'Under 11s'),
        ... #list other team levels
    )
    team_level =
models.CharField(max_length=3, choices=TEAM_LEVELS, default='U11')

```

Note: A little bit of Python:

- Python supports "object-oriented programming", a style of programming where we organise our code into objects, which include related data and functions for operating on that data. Objects can also inherit/extend/derive from other objects, allowing common behaviour between related objects to be shared. In Python we use the keyword `class` to define the "blueprint" for an object. We can create multiple specific *instances* of the type of object based on the model in the class.

So for example, here we have a `Team` class, which derives from the `Model` class. This means it IS a model, and will contain all the methods of a model, but we can also give it specialized features of its own too. In our model we define the fields our database will need to store our data, giving them specific names. Django uses these definitions, including the field names, to create the underlying database.

Querying data (views.py)

The Django model provides a simple query API for searching the database. This can match against a number of fields at a time using different criteria (e.g. exact, case-insensitive, greater than, etc.), and can support complex statements (for example, you can specify a search on U11 teams that have a team name that starts with "Fr" or ends with "al").

The code snippet shows a view function (resource handler) for displaying all of our U09 teams. The line in bold shows how we can use the model query API to filter for all records where the `team_level` field has exactly the text 'U09' (note how this criteria is passed to the `filter()` function as an argument with the field name and match type separated by a double underscore: **`team_level__exact`**).

```
## filename: views.py

from django.shortcuts import render
from .models import Team

def index(request):
    list_teams = Team.objects.filter(team_level__exact="U09")
    context = {'youngest_teams': list_teams}
    return render(request, '/best/index.html', context)
```

This function uses the `render()` function to create the `HttpResponse` that is sent back to the browser. This function is a *shortcut*; it creates an HTML file by combining a specified HTML template and some data to insert in the template (provided in the variable named "context"). In the next section we show how the template has the data inserted in it to create the HTML.

Rendering data (HTML templates)

Template systems allow you to specify the structure of an output document, using placeholders for data that will be filled in when a page is generated. Templates are often used to create HTML, but can also create other types of document. Django supports both its native templating system and another popular Python library called Jinja2 out of the box (it can also be made to support other systems if needed).

The code snippet shows what the HTML template called by the `render()` function in the previous section might look like. This template has been written under the assumption that it will have access to a list variable called `youngest_teams` when it is rendered (contained in the context variable inside the `render()` function above). Inside the HTML skeleton we have an expression that first checks if the `youngest_teams` variable exists, and then iterates it in a `for` loop. On each iteration the template displays each team's `team_name` value in an `` element.

```
## filename: best/templates/best/index.html

<!DOCTYPE html>
<html lang="en">
<body>

    {% if youngest_teams %}
        <ul>
            {% for team in youngest_teams %}
                <li>{{ team.team_name }}</li>
            {% endfor %}
        </ul>
    {% else %}
        <p>No teams are available.</p>
    {% endif %}

</body>
</html>
```

What else can you do?[Edit](#)

The preceding sections show the main features that you'll use in almost every web application: URL mapping, views, models and templates. Just a few of the other things provided by Django include:

- **Forms:** HTML Forms are used to collect user data for processing on the server. Django simplifies form creation, validation, and processing.
- **User authentication and permissions:** Django includes a robust user authentication and permission system that has been built with security in mind.
- **Caching:** Creating content dynamically is much more computationally intensive (and slow) than serving static content. Django provides flexible caching so that you can store all or part of a rendered page so that it doesn't get re-rendered except when necessary.
- **Administration site:** The Django administration site is included by default when you create an app using the basic skeleton. It makes it trivially easy to provide an admin page for site administrators to create, edit, and view any data models in your site.
- **Serialising data:** Django makes it easy to serialise and serve your data as XML or JSON. This can be useful when creating a web service (a web site that purely serves data to be consumed by other applications or sites, and doesn't display anything itself), or when creating a website in which the client-side code handles all the rendering of data.

Summary[Edit](#)

Congratulations, you've completed the first step in your Django journey! You should now understand Django's main benefits, a little about its history, and roughly what each of the main parts of a Django app might look like. You should have also learned a few things about the Python programming language, including the syntax for lists, functions, and classes.

You've already seen some real Django code above, but unlike with client-side code, you need to set up a development environment to run it. That's our next step.

Setting up a Django development environment

Now that you know what Django is for, we'll show you how to set up and test a Django development environment on Windows, Linux (Ubuntu), and Mac OS X — whatever common operating system you are using, this article should give you what you need to be able to start developing Django apps.

Prerequisites: Know how to open a terminal / command line. Know how to install software packages on your development computer's operating system.

Objective: To have a development environment for Django (1.10) running on your computer.

[Django development environment overview](#)[Edit](#)

Django makes it very easy to set up your own computer so that you can start developing web applications. This section explains what you get with the development environment, and provides an overview of some of your setup and configuration options. The remainder of the article explains the *recommended* method of installing the Django development environment on Ubuntu, Mac OS X, and Windows, and how you can test it.

[What is the Django development environment?](#)

The development environment is an installation of Django on your local computer that you can use for developing and testing Django apps prior to deploying them to a production environment.

The main tools that Django itself provides are a set of Python scripts for creating and working with Django projects, along with a simple *development webserver* that you can use to test local (i.e. on your computer, not on an external web server) Django web applications on your computer's web browser.

There are other peripheral tools, which form part of the development environment, that we won't be covering here. These include things like a [text editor](#) or IDE for editing code, and a source control management tool like [Git](#) for safely managing different versions of your code. We are assuming that you've already got a text editor installed.

What are the Django setup options?

Django is extremely flexible in terms of how and where it can be installed and configured. Django can be:

- installed on different operating systems.
- used with Python 3 and Python 2.
- installed from source, from the Python Package Index (PyPi) and in many cases from the host computer's package manager application.
- configured to use one of several databases, which may also need to be separately installed and configured.
- run in the main system Python environment or within separate Python virtual environments.

Each of these options requires slightly different configuration and setup. The following subsections explain some of your choices. For the rest of the article we'll show you how to setup Django on a small number of operating systems, and that setup will be assumed throughout the rest of this module.

Note: Other possible installation options are covered in the official Django documentation. We link to the [appropriate documents below](#).

What operating systems are supported?

Django web applications can be run on almost any machine that can run the Python programming language: Windows, Mac OS X, Linux/Unix, Solaris, to name just a few. Almost any computer should have the necessary performance to run Django during development.

In this article we'll provide instructions for Windows, Mac OS X, and Linux/Unix.

What version of Python should you use?

Django runs on top of Python, and can be used with either Python 2 or Python 3 (or both). When selecting a version you should be aware that:

- Python 2 is a legacy version of the language that is no longer receiving new features but has a truly enormous repository of high quality 3rd party libraries for developers to use (some of which are not available for Python 3).
- Python 3 is an update of Python 2 that, while similar, is more consistent and easier to use. Python 3 is also the future of Python, and continues to evolve.
- It is also possible to support both versions using compatibility libraries (e.g. [six](#)), although not without additional development effort.

Note: Historically Python 2 was the only realistic choice, because very few 3rd party libraries were available for Python 3. The current trend is that most new and popular packages on the [Python Package Index](#) (PyPi) support both versions of Python. While there are still many packages that are only available for Python 2, choosing Python 3 is now a feasible option.

We recommend that you use the latest version of Python 3 unless the site depends on 3rd party libraries that are only available for Python 2.

This article will explain how to install an environment for Python 3 (the equivalent setup for Python 2 would be very similar).

Where can we download Django?

There are three places to download Django:

- The Python Package Repository (PyPi), using the *pip* tool. This is the best way to get the latest stable version of Django.
- Use a version from your computer's package manager. Distributions of Django that are bundled with operating systems offer a familiar installation mechanism. Note however that the packaged version may be quite old, and can only be installed into the system Python environment (which may not be what you want).
- Install from source. You can get and install the latest bleeding-edge version of Python from source. This is not recommended for beginners, but is needed when you're ready to start contributing back to Django itself.

This article shows how to install Django from PyPi, in order to get the latest stable version.

Which database?

Django supports four main databases (PostgreSQL, MySQL, Oracle and SQLite), and there are community libraries that provide varying levels of support for other popular SQL and NOSQL databases. We recommend that you select the same database for both production and development (although Django abstracts many of the database differences using its Object-Relational Mapper (ORM), there are still [potential issues](#) that are better to avoid).

For this article (and most of this module) we will be using the *SQLite* database, which stores its data in a file. SQLite is intended for use as a lightweight database and can't support a high level of concurrency. It is however an excellent choice for applications that are primarily read-only.

Note: Django is configured to use SQLite by default when you start your website project using the standard tools (*django-admin*). It's a great choice when you're getting started because it requires no additional configuration or setup.

Installing system-wide or in a Python virtual environment?

When you install Python 3 on your computer you get a single global environment (set of installed packages) for your Python code, which you manage using the *pip3* tool. While you can install whatever Python packages you like in this environment, you can only install one particular version at a time. This means that any changes you make to any Python application can potentially affect all others, and you can only have one Django environment/version at a time.

Experienced Python/Django developers often choose to instead run their Python apps within independent *Python virtual environments*. These allow developers to have multiple different Django environments on a single computer, allowing them to create new websites (using the latest version of Django) while still maintaining websites that rely on older versions. The Django developer team itself recommends that you use Python virtual environments!

When you're getting started the approach you use doesn't particularly matter. As the setup is a little easier, we've decided to show you how to install Django directly into the system-wide Python 3 environment.

Important: The rest of this article shows how to set up Django into the system-wide Python 3 environment, on Ubuntu Linux, Mac OS X, and Windows 10.

Installing Python 3 [Edit](#)

In order to use Django you will have to install *Python 3* on your operating system. You will also need the [Python Package Index](#) tool — *pip3* — which is used to manage (install, update, and remove) Python packages/libraries used by Django and your other Python apps.

This section briefly explains how you can check what versions are present, and install new versions as needed, for Ubuntu Linux 16.04, Mac OS X, and Windows 10.

Note: Depending on your platform, you may also be able to install Python/pip from the operating system's own package manager or via other mechanisms. For most platforms you can download the required installation files from <https://www.python.org/downloads/> and install them using the appropriate platform-specific method.

Ubuntu 16.04

Ubuntu Linux includes Python 3 by default. You can confirm this by running the following command in the bash terminal:

```
python3 -V
Python 3.5.2
```

However the Python Package Index tool you'll need to install packages for Python 3 (including Django) is **not** available by default. You can install pip3 in the bash terminal using:

```
sudo apt-get install python3-pip
```

Mac OS X

Mac OS X "El Capitan" does not include Python 3. You can confirm this by running the following commands in the bash terminal:

```
python3 -V
-bash: python3: command not found
```

You can easily install Python 3 (along with the *pip3* tool) from [python.org](https://www.python.org/):

1. Download the required installer:
 1. Go to <https://www.python.org/downloads/>
 2. Select the **Download Python 3.5.2** button (the exact minor version number may differ).
2. Locate the file using *Finder*, and double-click the package file. Following the installation prompts.

You can now confirm successful installation by checking for the *Python 3* as shown below:

```
python3 -V
Python 3.5.20
```

You can similarly check that *pip3* is installed by listing the available packages:

```
pip3 list
Windows 10
```

Windows doesn't include Python by default, but you can easily install it (along with the *pip3* tool) from [python.org](https://www.python.org):

1. Download the required installer:
 1. Go to <https://www.python.org/downloads/>
 2. Select the **Download Python 3.5.2** button (the exact minor version number may differ).
2. Install Python by double-clicking on the downloaded file and following the installation prompts

You can then verify that Python was installed by entering the following text into the command prompt:

```
py -3 -V
Python 3.5.2
```

The Windows installer incorporates *pip3* (the Python package manager) by default. You can list installed packages as shown:

```
pip3 list
Installing Django
```

Once you've installed *Python 3* and *pip3*, you can use *pip3* to install Django.

```
pip3 install django
```

You can test that Django is installed by running the following command (this just tests that Python can find the Django module):

```
# Linux/Mac OS X
python3 -m django --version
1.10.10

# Windows
py -3 -m django --version
1.10.10
```

Note: On Windows you launch *Python 3* scripts by prefixing the command with `py -3`, while on Linux/Mac OSX, the command is `python3`.

Important: The rest of this **module** uses the *Linux* command for invoking Python 3 (`python3`). If you're working on *Windows* simply replace this prefix with: `py -3`

Testing your installation[Edit](#)

The above test works, but it isn't very much fun. A more interesting test is to create a skeleton project and see it working. To do this, first navigate in your command prompt/terminal to where you want to store you Django apps. Create a folder for your test site and navigate into it.

```
mkdir django_test
cd django_test
```

You can then create a new skeleton site called "*mytestsite*" using the **django-admin** tool as shown. After creating the site you can navigate into the folder where you will find the main script for managing projects, called **manage.py**.

```
django-admin startproject mytestsite
cd mytestsite
```

We can run the *development web server* from within this folder using **manage.py** and the **runserver** command, as shown.

```
$ python3 manage.py runserver
Performing system checks...
```

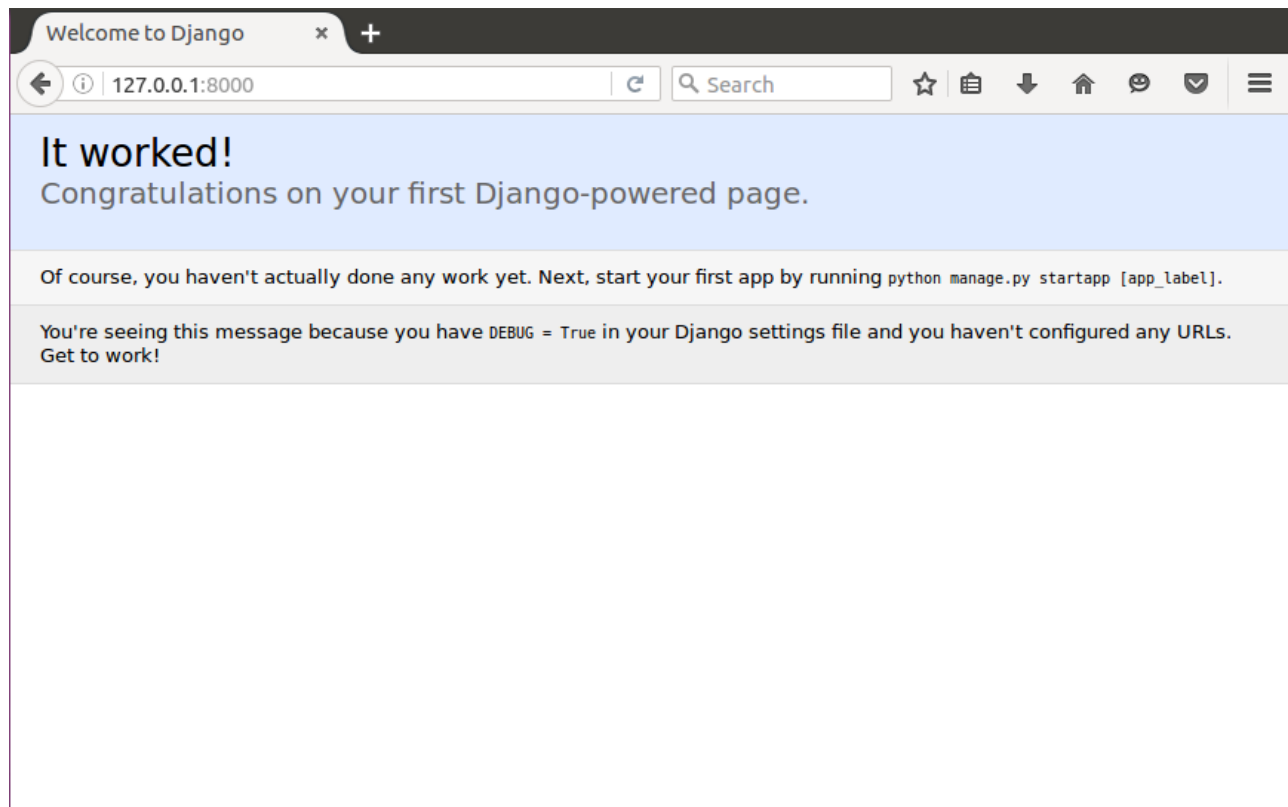
```
System check identified no issues (0 silenced).
```

```
You have 13 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
```

```
September 19, 2016 - 23:31:14
Django version 1.10.1, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Note: The above command shows the Linux/Mac OS X command. You can ignore the warnings about "13 unapplied migration(s)" at this point!

Once the server is running you can view the site by navigating to the following URL on your local web browser: <http://127.0.0.1:8000/>. You should see a site that looks like this:



[Summary](#)[Edit](#)

You now have a Django development environment up and running on your computer.

In the testing section you also briefly saw how we can create a new Django website using `django-admin startproject`, and run it in your browser using the development web server (`python3 manage.py runserver`). In the next article we expand on this process, building a simple but complete web application.

Django Tutorial: The Local Library website

The first article in our practical tutorial series explains what you'll learn, and provides an overview of the "local library" example website we'll be working through and evolving in subsequent articles.

Prerequisites: Read the [Django Introduction](#). For the following articles you'll also need to have [set up a Django development environment](#).

Objective: To introduce the example application used in this tutorial, and allow readers to understand what topics will be covered.

[Overview](#)[Edit](#)

Welcome to the MDN "Local Library" Django tutorial, in which we develop a website that might be used to manage the catalog for a local library.

In this series of tutorial articles you will:

- Use Django's tools to create a skeleton website and application.
- Start and stop the development server.
- Create models to represent your application's data.
- Use the Django admin site to populate your site's data.
- Create views to retrieve specific data in response to different requests, and templates to render the data as HTML to be displayed in the browser.
- Create mappers to associate different URL patterns with specific views.
- Add user authorisation and sessions to control site behaviour and access.
- Work with forms.
- Write test code for your app.
- Use Django's security effectively.
- Deploy your application to production.

You have learnt about some of these topics already, and touched briefly on others. By the end of the tutorial series you should know enough to develop simple Django apps by yourself.

[The LocalLibrary website](#)[Edit](#)

LocalLibrary is the name of the website that we'll create and evolve over the course of this series of tutorials. As you'd expect, the purpose of the website is to provide an online catalog for a small local library, where users can browse available books and manage their accounts.

This example has been carefully chosen because it can scale to show as much or little detail as we need, and can be used to show off almost any Django feature. More importantly, it allows us to provide a *guided* path through the most important functionality in the Django web framework:

- In the first few tutorial articles we will define a simple *browse-only* library that library members can use to find out what books are available. This allows us to explore the operations that are common to almost every website: reading and displaying content from a database.
- As we progress, the library example naturally extends to demonstrate more advanced Django features. For example we can extend the library to allow users to reserve books, and use this to demonstrate how to use forms, and support user authentication.

Even though this is a very extensible example, it's called ***LocalLibrary*** for a reason — we're hoping to show the minimum information that will help you get up and running with Django quickly. As a result we'll store information about books, copies of books, authors and other key

information. We won't however be storing information about other items a library might store, or provide the infrastructure needed to support multiple library sites or other "big library" features.

I'm stuck, where can I get the source?[Edit](#)

As you work through the tutorial we'll provide the appropriate code snippets for you to copy and paste at each point, and there will be other code that we hope you'll extend yourself (with some guidance).

If you get stuck, you can find the fully developed version of the website [on Github here](#).

Summary[Edit](#)

Now that you know a bit more about the *LocalLibrary* website and what you're going to learn, it's time to start creating a [skeleton project](#) to contain our example.

Django Tutorial Part 2: Creating a skeleton website

This second article in our [Django Tutorial](#) shows how you can create a "skeleton" website project as a basis, which you can then go on to populate with site-specific settings, urls, models, views, and templates.

Prerequisites: [Set up a Django development environment](#). Review the [Django Tutorial](#).

Objective: To be able to use Django's tools to start your own new website projects.

[Overview](#)[Edit](#)

This article shows how you can create a "skeleton" website, which you can then populate with site-specific settings, urls, models, views, and templates (we discuss these in later articles).

The process is straightforward:

1. Use the `django-admin` tool to create the project folder, basic file templates, and project management script (**manage.py**).
2. Use **manage.py** to create one or more *applications*.

Note: A website may consist of one or more sections, e.g. main site, blog, wiki, downloads area, etc. Django encourages you to develop these components as separate *applications*, which could then be re-used in different projects if desired.

3. Register the new applications to include them in the project.
4. Hook up the url mapper for each application.

For the [Local Library website](#) the website folder and its project folder will be named *locallibrary*, and we'll have just one application named *catalog*. The top level folder structure will therefore be as follows:

```
locallibrary/           # Website folder
    manage.py          # Script to run Django tools for this project (created
using django-admin)
    locallibrary/      # Website/project folder (created using django-admin)
    catalog/           # Application folder (created using manage.py)
```

The following sections discuss the process steps in detail, and show how you can test the changes. At the end of the article we discuss some of the other site-wide configuration you might also do at this stage.

[Creating the project](#)[Edit](#)

First open a command prompt/terminal, navigate to where you want to store your Django apps (make it somewhere easy to find like inside your *documents* folder), and create a folder for your new website (in this case: *locallibrary*). Then enter into the folder using the `cd` command:

```
mkdir locallibrary
cd locallibrary
```

Create the new project using the `django-admin startproject` command as shown, and then navigate into the folder.

```
django-admin startproject locallibrary
cd locallibrary
```

The `django-admin` tool creates a folder/file structure as shown below:

```
locallibrary/
  manage.py
  locallibrary/
    settings.py
    urls.py
    wsgi.py
```

The *locallibrary* project sub-folder is the entry point for the website:

- **settings.py** contains all the website settings. This is where we register any applications we create, the location of our static files, database configuration details, etc.
- **urls.py** defines the site url-to-view mappings. While this could contain *all* the url mapping code, it is more common to delegate some of the mapping to particular applications, as you'll see later.
- **wsgi.py** is used to help your Django application communicate with the web server. You can treat this as boilerplate.

The **manage.py** script is used to create applications, work with databases, and start the development web server.

Creating the catalog application [Edit](#)

Next, run the following command to create the *catalog* application that will live inside our localibrary project (this must be run in the same folder as your project's **manage.py**):

```
python3 manage.py startapp catalog
```

Note: the above command is for Linux/Mac OS X. On Windows the command should be: `py -3 manage.py startapp catalog`

If you're working on Windows, make the replacement of `python3` with `py -3` throughout this module.

The tool creates a new folder and populates it with files for the different parts of the application (shown in bold below). Most of the files are usefully named after their purpose (e.g. views should be stored in **views.py**, Models in **models.py**, tests in **tests.py**, administration site configuration in **admin.py**, application registration in **apps.py**) and contain some minimal boilerplate code for working with the associated objects.

The updated project directory should now look like this:

```
locallibrary/  
  manage.py  
  locallibrary/  
  catalog/  
    admin.py  
    apps.py  
    models.py  
    tests.py  
    views.py  
    __init__.py  
    migrations/
```

In addition we now have:

- A *migrations* folder, used to store "migrations" — files that allow you to automatically update your database as you modify your models.
- **`__init__.py`** — an empty file created here so that Django/Python will recognise the folder as a [Python Package](#) and allow you to use its objects within other parts of the project.

Note: Have you noticed what is missing from the files list above? While there is a place for your views and models, there is nowhere for you to put your url mappings, templates, and static files. We'll show you how to create them further along (these aren't needed in every website but they are needed in this example).

Registering the catalog application [Edit](#)

Now that the application has been created we have to register it with the project so that it will be included when any tools are run (for example to add models to the database). Applications are registered by adding them to the `INSTALLED_APPS` list in the project settings.

Open the project settings file **`locallibrary/locallibrary/settings.py`** and find the definition for the `INSTALLED_APPS` list. Then add a new line at the end of the list, as shown in bold below.

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

The new line specifies the application configuration object (`CatalogConfig`) that was generated for you in **`locallibrary/catalog/apps.py`** when you created the application.

Note: You'll notice that there are already a lot of other `INSTALLED_APPS` (and `MIDDLEWARE`, further down in the settings file). These enable support for the [Django administration site](#) and as a result lots of the functionality it uses (including sessions, authentication, etc).

Specifying the database[Edit](#)

This is also the point where you would normally specify the database to be used for the project — it makes sense to use the same database for development and production where possible, in order to avoid minor differences in behaviour. You can find out about the different options in [Databases](#) (Django docs).

We'll use the SQLite database for this example, because we don't expect to require a lot of concurrent access on a demonstration database, and also because it requires no additional work to set up! You can see how this database is configured in **settings.py** (more information is also included below):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Because we are using SQLite, we don't need to do any further setup here. Let's move on!

Other project settings[Edit](#)

The **settings.py** file is also used for configuring a number of other settings, but at this point you probably only want to change the [TIME_ZONE](#) — this should be made equal to a string from the standard [List of tz database time zones](#) (the TZ column in the table contains the values you want). Change your `TIME_ZONE` value to one of these strings appropriate for your time zone, for example:

```
TIME_ZONE = 'Europe/London'
```

There are two other settings you won't change now, but that you should be aware of:

- `SECRET_KEY`. This is a secret key that is used as part of Django's website security strategy. If you're not protecting this code in development, you'll need to use a different code (perhaps read from an environment variable or file) when putting it into production.
- `DEBUG`. This enables debugging logs to be displayed on error, rather than HTTP status code responses. This should be set to `false` on production as debug information is useful for attackers.

Hooking up the URL mapper[Edit](#)

The website is created with a URL mapper file (**urls.py**) in the project folder. While you can use this file to manage all your URL mappings, it is more usual to defer mappings to their associated application.

Open **locallibrary/locallibrary/urls.py** and note the instructional text which explains some of the ways to use the URL mapper.

```
"""
locallibrary URL Configuration
```


The `urlpatterns` list routes URLs to views. For more information please see:
<https://docs.djangoproject.com/en/1.10/topics/http/urls/>

Examples:

Function views

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

Class-based views

1. Add an import: `from other_app.views import Home`
2. Add a URL to `urlpatterns`: `url(r'^$', Home.as_view(), name='home')`

Including another `URLconf`

1. Import the `include()` function: `from django.conf.urls import url, include`

2. Add a URL to `urlpatterns`: `url(r'^blog/', include('blog.urls'))`

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

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

The URL mappings are managed through the `urlpatterns` variable, which is a Python *list* of `url()` functions. Each `url()` function either associates a URL pattern to a *specific view* or with another list of URL pattern testing code (in this second case, the pattern becomes the "base URL" for patterns defined in the target module). The `urlpatterns` list initially defines a single function that maps all URLs with the pattern `admin/` to the module `admin.site.urls`, which contains the Administration application's own URL mapping definitions.

Add the lines below to the bottom of the file in order to add a new list item to the `urlpatterns` list. This new item includes a `url()` that forwards requests with the pattern `catalog/` to the module `catalog.urls` (the file with the relative URL `/catalog/urls.py`).

```
# Use include() to add URLs from the catalog application
from django.conf.urls import include
```

```
urlpatterns += [
    url(r'^catalog/', include('catalog.urls')),
]
```

Now let's redirect the root URL of our site (i.e. `127.0.0.1:8000`) to the URL `127.0.0.1:8000/catalog/`; this is the only app we'll be using in this project, so we might as well. To do this, we'll use a special view function (`RedirectView`), which takes as its first argument the new relative URL to redirect to (`/catalog/`) when the URL pattern specified in the `url()` function is matched (the root URL, in this case).

Add the following lines, again to the bottom of the file:

```
#Add URL maps to redirect the base URL to our application
from django.views.generic import RedirectView
urlpatterns += [
    url(r'^$', RedirectView.as_view(url='/catalog/', permanent=True)),
]
```

Django does not serve static files like CSS, JavaScript, and images by default, but it can be useful for the development web server to do so while you're creating your site. As a final addition to this URL mapper, you can enable the serving of static files during development by appending the following lines.

Add the following final block to the bottom of the file now:

```
# Use static() to add url mapping to serve static files during development
(only)
from django.conf import settings
from django.conf.urls.static import static

urlpatterns += static(settings.STATIC_URL,
document_root=settings.STATIC_ROOT)
```

Note: There are a number of ways to extend the `urlpatterns` list (above we just appended a new list item using the `+=` operator to clearly separate the old and new code). We could have instead just included this new pattern-map in the original list definition:

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^catalog/', include('catalog.urls')),
    url(r'^$', RedirectView.as_view(url='/catalog/', permanent=True)),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

In addition, we included the import line (`from django.conf.urls import include`) down here so it is easy to see what we've added, but it is common to include all your import lines at the top of a Python file.

As a final step, create a file inside your catalog folder called **urls.py**, and add the following text to define the (empty) imported `urlpatterns`. This is where we'll add our patterns as we build the application.

```
from django.conf.urls import url

from . import views

urlpatterns = [

]
```

[Testing the website framework](#)[Edit](#)

At this point we have a complete skeleton project. The website doesn't actually *do* anything yet, but its worth running it to make sure that none of our changes have broken anything.

Before we do that, we should first run a *database migration*. This updates our database to include any models in our installed applications (and removes some build warnings).

[Running database migrations](#)

Django uses an Object-Relational-Mapper (ORM) to map Model definitions in the Django code to the data structure used by the underlying database. As we change our model definitions,

Django tracks the changes and can create database migration scripts (in `/locallibrary/catalog/migrations/`) to automatically migrate the underlying data structure in the database to match the model.

When we created the website Django automatically added a number of models for use by the admin section of the site (which we'll look at later). Run the following commands to define tables for those models in the database (make sure you are in the directory that contains `manage.py`):

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Important: You'll need to run the above commands every time your models change in a way that will affect the structure of the data that needs to be stored (including both addition and removal of whole models and individual fields).

The `makemigrations` command *creates* (but does not apply) the migrations for all applications installed in your project (you can specify the application name as well to just run a migration for a single project). This gives you a chance to checkout the code for these migrations before they are applied — when you're a Django expert you may choose to tweak them slightly!

The `migrate` command actually applies the migrations to your database (Django tracks which ones have been added to the current database).

Note: See [Migrations](#) (Django docs) for additional information about the lesser-used migration commands.

Running the website

During development you can test the website by first serving it using the *development web server*, and then viewing it on your local web browser.

Note: The development web server is not robust or performant enough for production use, but it is a very easy way to get your Django website up and running during development to give it a convenient quick test. By default it will serve the site to your local computer (`http://127.0.0.1:8000/`), but you can also specify other computers on your network to serve to. For more information see [django-admin and manage.py: runserver](#) (Django docs).

Run the *development web server* by calling the `runserver` command (in the same directory as `manage.py`):

```
python3 manage.py runserver
```

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

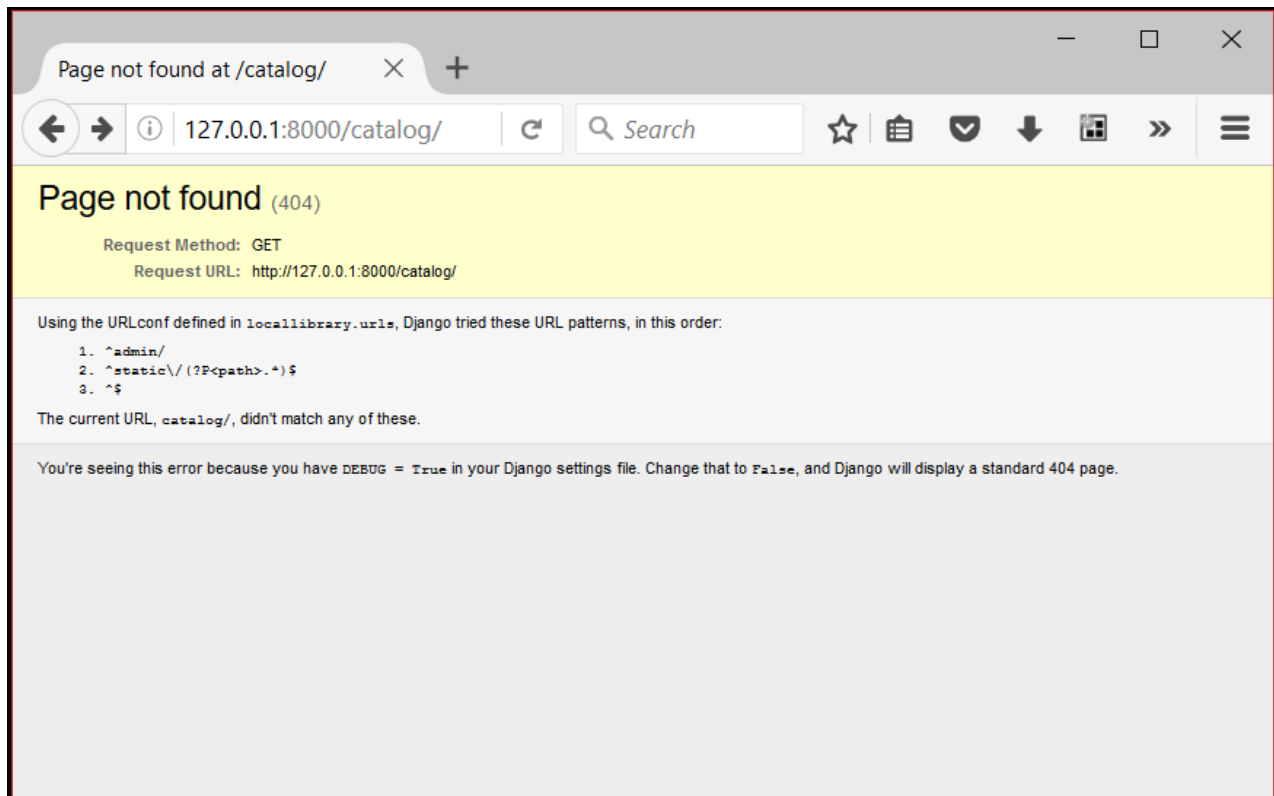
```
September 22, 2016 - 16:11:26
```

```
Django version 1.10, using settings 'locallibrary.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CTRL-BREAK.
```

Once the server is running you can view the site by navigating to `http://127.0.0.1:8000/` in your local web browser. You should see a site error page that looks like this:



Don't worry! This error page is expected because we don't have any pages/urls defined in the `catalogs.urls` module (which we're redirected to when we get an URL to the root of the site).

Note: The above page demonstrates a great Django feature — automated debug logging. An error screen will be displayed with useful information whenever a page cannot be found, or any error is raised by the code. In this case we can see that the URL we've supplied doesn't match any of our URL patterns (as listed). The logging will be turned off during production (when we put the site live on the Web), in which case a less informative but more user-friendly page will be served.

At this point we know that Django is working!

Note: You should re-run migrations and re-test the site whenever you make significant changes. It doesn't take very long!

Challenge yourself [Edit](#)

The **catalog/** directory contains files for the views, models, and other parts of the application. Open these files and inspect the boilerplate.

As you saw above, a URL-mapping for the Admin site has already been added in the project's **urls.py**. Navigate to the admin area in your browser and see what happens (you can infer the correct URL from the mapping above).

[Summary](#)[Edit](#)

You have now created a complete skeleton website project, which you can go on to populate with urls, models, views, and templates.

Now the skeleton for the [Local Library website](#) is complete and running, it's time to start writing the code that makes this website do what it is supposed to do.

Django Tutorial Part 3: Using models

This article shows how to define models for the [LocalLibrary](#) website. It explains what a model is, how it is declared, and some of the main field types. It also briefly shows a few of the main ways you can access model data.

Prerequisites: [Django Tutorial Part 2: Creating a skeleton website](#).

Objective: To be able to design and create your own models, choosing fields appropriately.

Overview[Edit](#)

Django web applications access and manage data through Python objects referred to as models. Models define the *structure* of stored data, including the field *types* and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.

This tutorial shows how to define and access the models for the [LocalLibrary website](#) example.

Designing the LocalLibrary models[Edit](#)

Before you jump in and start coding the models, it's worth taking a few minutes to think about what data we need to store and the relationships between the different objects.

We know that we need to store information about books (title, summary, author, written language, category, ISBN) and that we might have multiple copies available (with globally unique id, availability status, etc.). We might need to store more information about the author than just their name, and there might be multiple authors with the same or similar names. We want to be able to sort information based on book title, author, written language, and category.

When designing your models it makes sense to have separate models for every "object" (group of related information). In this case the obvious objects are books, book instances and authors.

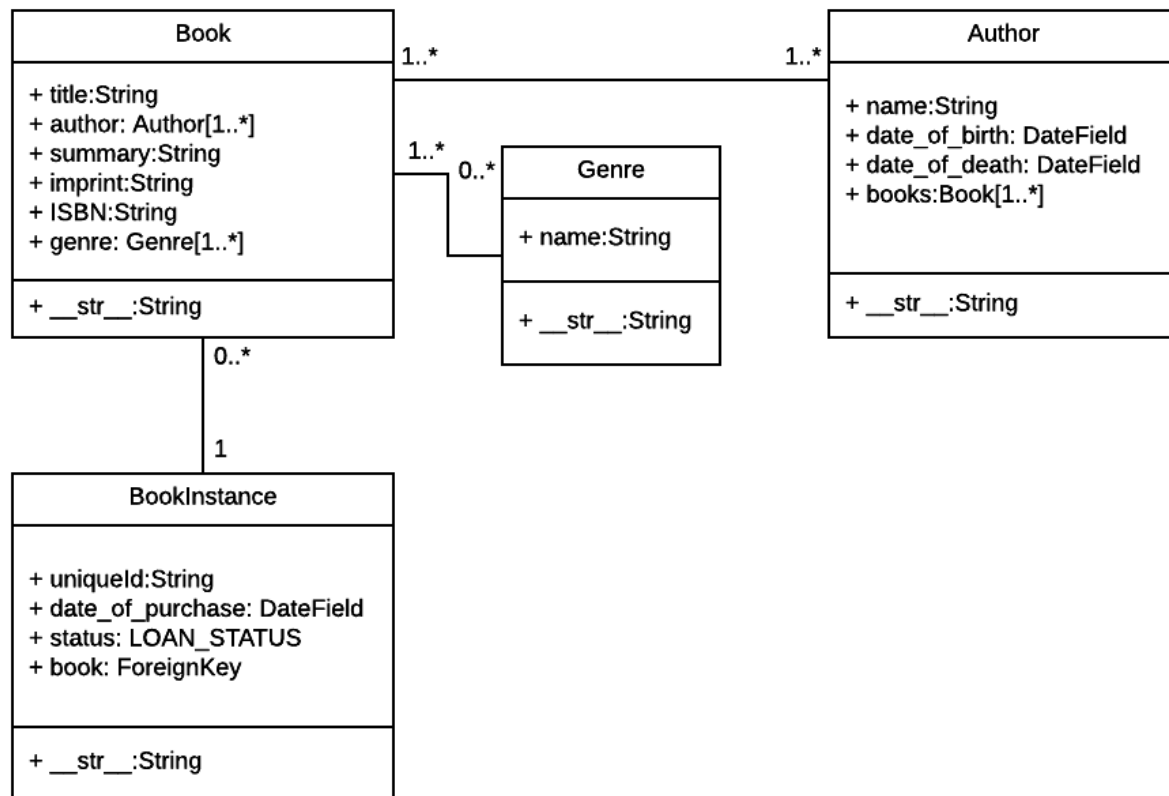
You might also want to use models to represent selection-list options (e.g. like a drop down list of choices), rather than hard coding the choices into the website itself — this is recommended when all the options aren't known up front or may change. Obvious candidates for models in this case include the book genre (e.g. Science Fiction, French Poetry, etc.) and language (English, French, Japanese).

Once we've decided on our models and field, we need to think about the relationships. Django allows you to define relationships that are one to one (`OneToOneField`), one to many (`ForeignKey`) and many to many (`ManyToManyField`).

With that in mind, the UML association diagram below shows the models we'll define in this case (as boxes). As above, we've created models for book (the generic details of the book), book

instance (status of specific physical copies of the book available in the system), and author. We have also decided to have a model for genre, so that values can be created/selected through the admin interface. We've decided not to have a model for the `BookInstance:status` — we've hard coded the values (`LOAN_STATUS`) because we don't expect these to change. Within each of the boxes you can see the model name, the field names and types, and also the methods and their return types.

The diagram also shows the relationships between the models, including their *multiplicities*. The multiplicities are the numbers on the diagram showing the numbers (maximum and minimum) of each model that may be present in the relationship. For example, the connecting line between the boxes shows that Book and a Genre are related. The numbers close to the Book model show that a book must have one or more Genres (as many as you like) , while the numbers on the other end of the line next to the Genre show that it can have zero or more associated books.



Note: The next section provides a basic primer explaining how models are defined and used. As you read it, consider how we will construct each of the models in the diagram above.

[Model primer](#)[Edit](#)

This section provides a brief overview of how a model is defined and some of the more important fields and field arguments.

Model definition

Models are usually defined in an application's **models.py** file. They are implemented as subclasses of `django.db.models.Model`, and can include fields, methods and metadata. The code fragment below shows a "typical" model, named `MyModelName`:

```
from django.db import models

class MyModelName(models.Model):
    """
    A typical class defining a model, derived from the Model class.
    """

    # Fields
    my_field_name = models.CharField(max_length=20, help_text="Enter field
documentation")
    ...

    # Metadata
    class Meta:
        ordering = ["-my_field_name"]

    # Methods
    def get_absolute_url(self):
        """
        Returns the url to access a particular instance of MyModelName.
        """
        return reverse('model-detail-view', args=[str(self.id)])

    def __str__(self):
        """
        String for representing the MyModelName object (in Admin site etc.)
        """
        return self.field_name
```

In the below sections we'll explore each of the features inside the model in detail:

Fields

A model can have an arbitrary number of fields, of any type — each one represents a column of data that we want to store in one of our database tables. Each database record (row) will consist of one of each field value. Let's look at the example seen above:

```
my_field_name = models.CharField(max_length=20, help_text="Enter field
documentation")
```

Our above example has a single field called `my_field_name`, of type `models.CharField` — which means that this field will contain strings of alphanumeric characters. The field types are assigned using specific classes, which determine the type of record that is used to store the data in the database, along with validation criteria to be used when values are received from an HTML form (i.e. what constitutes a valid value). The field types can also take arguments that further specify how the field is stored or can be used. In this case we are giving our field two arguments:

- `max_length=20` — States that the maximum length of a value in this field is 20 characters.

- `help_text="Enter field documentation"` — provides a text label to display to help users know what value to provide when this value is to be entered by a user via an HTML form.

The field name is used to refer to it in queries and templates. Fields also have a label, which is either specified as an argument (`verbose_name`) or inferred by capitalising the first letter of the field's variable name and replacing any underscores with a space (for example `my_field_name` would have a default label of *My field name*).

The order that fields are declared will affect their default order if a model is rendered in a form (e.g. in the Admin site), though this may be overridden.

Common field arguments

The following common arguments can be used when declaring many/most of the different field types:

- [help_text](#): Provides a text label for HTML forms (e.g. in the admin site), as described above.
- [verbose_name](#): A human-readable name for the field used in field labels. If not specified, Django will infer the default verbose name from the field name.
- [default](#): The default value for the field. This can be a value or a callable object, in which case the object will be called every time a new record is created.
- [null](#): If `True`, Django will store blank values as `NULL` in the database for fields where this is appropriate (a `CharField` will instead store an empty string). The default is `False`.
- [blank](#): If `True`, the field is allowed to be blank in your forms. The default is `False`, which means that Django's form validation will force you to enter a value. This is often used with `null=True`, because if you're going to allow blank values, you also want the database to be able to represent them appropriately.
- [choices](#): A group of choices for this field. If this is provided, the default corresponding form widget will be a select box with these choices instead of the standard text field.
- [primary_key](#): If `True`, sets the current field as the primary key for the model (A primary key is a special database column designated to uniquely identify all the different table records). If no field is specified as the primary key then Django will automatically add a field for this purpose.

There are many other options — you can view the [full list of field options here](#).

Common field types

The following list describes some of the more commonly used types of fields.

- [CharField](#) is used to define short-to-mid sized fixed-length strings. You must specify the `max_length` of the data to be stored.
- [TextField](#) is used for large arbitrary-length strings. You may specify a `max_length` for the field, but this is used only when the field is displayed in forms (it is not enforced at the database level).
- [IntegerField](#) is a field for storing integer (whole number) values, and for validating entered values as integers in forms.
- [DateField](#) and [DateTimeField](#) are used for storing/representing dates and date/time information (as Python `datetime.date` in and `datetime.datetime` objects, respectively). These fields can additionally declare the (mutually exclusive) parameters `auto_now=True` (to set the field to the current date every time the model is saved), `auto_now_add` (to only set the date when

the model is first created) , and `default` (to set a default date that can be overridden by the user).

- [EmailField](#) is used to store and validate email addresses.
- [FileField](#) and [ImageField](#) are used to upload files and images respectively (the `ImageField` simply adds additional validation that the uploaded file is an image). These have parameters to define how and where the uploaded files are stored.
- [AutoField](#) is a special type of `IntegerField` that automatically increments. A primary key of this type is automatically added to your model if you don't explicitly specify one.
- [ForeignKey](#) is used to specify a one-to-many relationship to another database model (e.g. a car has one manufacturer, but a manufacturer can make many cars). The "one" side of the relationship is the model that contains the key.
- [ManyToManyField](#) is used to specify a many-to-many relationship (e.g. a book can have several genres, and each genre can contain several books). In our library app we will use these very similarly to `ForeignKeys`, but they can be used in more complicated ways to describe the relationships between groups. These have the parameter `on_delete` to define what happens when the associated record is deleted (e.g. a value of `models.SET_NULL` would simply set the value to `NULL`).

There are many other types of fields, including fields for different types of numbers (big integers, small integers, floats), booleans, URLs, slugs, unique ids, and other "time-related" information (duration, time, etc.). You can view the [full list here](#).

Metadata

You can declare model-level metadata for your Model by declaring `class Meta`, as shown.

```
class Meta:
    ordering = ["-my_field_name"]
    ...
```

One of the most useful features of this metadata is to control the *default ordering* of records returned when you query the model type. You do this by specifying the match order in a list of field names to the `ordering` attribute, as shown above. The ordering will depend on the type of field (character fields are sorted alphabetically, while date fields are sorted in chronological order). As shown above, you can prefix the field name with a minus symbol (-) to reverse the sorting order.

So as an example, if we chose to sort books like this by default:

```
ordering = ["title", "-pubdate"]
```

the books would be sorted alphabetically by title, from A-Z, and then by publication date inside each title, from newest to oldest.

Another common attribute is `verbose_name`, a verbose name for the class in singular and plural form:

```
verbose_name = "BetterName"
```

Other useful attributes allow you to create and apply new "access permissions" for the model (default permissions are applied automatically), allow ordering based on another field, or to

declare that the class is "abstract" (a base class that you cannot create records for, and will instead be derived from to create other models).

Many of the other metadata options control what database must be used for the model and how the data is stored (these are really only useful if you need to map a model to an existing database).

The full list of metadata options are available here: [Model metadata options](#) (Django docs).

Methods

A model can also have methods.

Minimally, in every model you should define the standard Python class method `__str__()` to return a human-readable string for the each object. This string is used to represent individual records in the administration site (and anywhere else you need to refer to a model instance). Often this will return a title or name field from the model.

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

Another common method to include in Django models is `get_absolute_url()`, which returns a URL for displaying individual model records on the website (if you define this method then Django will automatically add a "View on Site" button to the model's record editing screens in the Admin site). A typical pattern for `get_absolute_url()` is shown below.

```
def get_absolute_url(self):
    """
    Returns the url to access a particular instance of the model.
    """
    return reverse('model-detail-view', args=[str(self.id)])
```

Note: Assuming you will use URLs like `/myapplication/mymodelname/2` to display individual records for your model (where "2" is the `id` for a particular record), you will need to create a URL mapper to pass the response and `id` to a "model detail view" (which will do the work required to display the record). The `reverse()` function above is able to "reverse" your url mapper (in the above case named `'model-detail-view'`) in order to create an URL of the right format.

Of course to make this work you still have to write the URL mapping, view, and template!

You can also define any other methods you like, and call them from your code or templates (provided that they don't take any parameters).

Model management

Once you've defined your model classes you can use them to create, update, or delete records, and to run queries to get all records or particular subsets of records. We'll show you how to do that in the tutorial when we define our views, but here is a brief summary.

Creating and modifying records

To create a record you can define an instance of the model and then call `save()`.

```
# Create a new record using the model's constructor.
a_record = MyModelName(my_field_name="Instance #1")

# Save the object into the database.
a_record.save()
```

Note: If you haven't declared any field as a `primary_key`, the new record will be given one automatically, with the field name `id`. You could query this field after saving the above record, and it would have a value of 1.

You can access the fields in this new record using the dot syntax, and change the values. You have to call `save()` to store modified values to the database.

```
# Access model field values using Python attributes.
print(a_record.id) #should return 1 for the first record.
print(a_record.my_field_name) # should print 'Instance #1'

# Change record by modifying the fields, then calling save().
a_record.my_field_name="New Instance Name"
a_record.save()
```

Searching for records

You can search for records that match a certain criteria using the model's `objects` attribute (provided by the base class).

Note: Explaining how to search for records using "abstract" model and field names can be a little confusing. In the discussion below we'll refer to a `Book` model with `title` and `genre` fields, where `genre` is also a model with a single field `name`.

We can get all records for a model as a `QuerySet`, using `objects.all()`. The `QuerySet` is an iterable object, meaning that it contains a number of objects that we can iterate/loop through.

```
all_books = Book.objects.all()
```

Django's `filter()` method allows us to filter the returned `QuerySet` to match a specified **text** or **numeric** field against a particular criteria. For example, to filter for books that contain "wild" in the title and then count them, we could do the following.

```
wild_books = Book.objects.filter(title__contains='wild')
number_wild_books = Book.objects.filter(title__contains='wild').count()
```

The fields to match and the type of match are defined in the filter parameter name, using the format: `field_name__match_type` (note the *double underscore* between `title` and `contains` above). Above we're filtering `title` with a case-sensitive match. There are many other types of matches you can do: `icontains` (case insensitive), `iexact` (case-insensitive exact match), `exact` (case-sensitive exact match) and `in`, `gt` (greater than), `startswith`, etc. The [full list is here](#).

In some cases you'll need to filter on a field that defines a one-to-many relationship to another model (e.g. a `ForeignKey`). In this case you can "index" to fields within the related model with additional double underscores. So for example to filter for books with a specific genre pattern, you will have to index to the `name` through the `genre` field, as shown below:

```
books_containing_genre =
Book.objects.filter(genre__name__icontains='fiction') # Will match on:
Fiction, Science fiction, non-fiction etc.
```

Note: You can use underscores (`__`) to navigate as many levels of relationships (`ForeignKey/ManyToManyField`) as you like. For example, a `Book` that had different types, defined using a further "cover" relationship might have a parameter name:

```
type__cover__name__exact='hard'.
```

There is a lot more you can do with queries, including backwards searches from related models, chaining filters, returning a smaller set of values etc. For more information see [Making queries](#) (Django Docs).

Defining the LocalLibrary Models [Edit](#)

In this section we will start defining the models for the library. Open *models.py* (in */locallibrary/catalog/*). The boilerplate at the top of the page imports the *models* module, which contains the model base class `models.Model` that our models will inherit from.

```
from django.db import models
```

```
# Create your models here.
```

Genre model

Copy the Genre model code shown below and paste it into the bottom of your *models.py* file. This model is used to store information about the book category — for example whether it is fiction or non-fiction, romance or military history, etc. As mentioned above, we've created the Genre as a model rather than as free text or a selection list so that the possible values can be managed through the database rather than being hard coded.

```
class Genre(models.Model):
    """
    Model representing a book genre (e.g. Science Fiction, Non Fiction).
    """
    name = models.CharField(max_length=200, help_text="Enter a book genre (e.g. Science Fiction, French Poetry etc.)")

    def __str__(self):
        """
        String for representing the Model object (in Admin site etc.)
        """
        return self.name
```

The model has a single `CharField` field (`name`), which is used to describe the genre (this is limited to 200 characters and has some `help_text`). At the end of the model we declare a `__str__()` method, which simply returns the name of the genre defined by a particular record. No verbose name has been defined, so the field will be called `Name` in forms.

Book model

Copy the Book model below and again paste it into the bottom of your file. The book model represents all information about an available book in a general sense, but not a particular physical "instance" or "copy" available for loan. The model uses a `CharField` to represent the book's title and `isbn` (note how the `isbn` specifies its label as "ISBN" using the first unnamed parameter because the default label would otherwise be "Isbn"). The model uses `TextField` for the `summary`, because this text may need to be quite long.

```
from django.urls import reverse #Used to generate URLs by reversing the URL
patterns

class Book(models.Model):
    """
    Model representing a book (but not a specific copy of a book).
    """
    title = models.CharField(max_length=200)
    author = models.ForeignKey('Author', on_delete=models.SET_NULL,
null=True)
    # Foreign Key used because book can only have one author, but authors can
have multiple books
    # Author as a string rather than object because it hasn't been declared
yet in the file.
    summary = models.TextField(max_length=1000, help_text="Enter a brief
description of the book")
    isbn = models.CharField('ISBN',max_length=13, help_text='13 Character <a
href="https://www.isbn-international.org/content/what-isbn">ISBN number</a>')
    genre = models.ManyToManyField(Genre, help_text="Select a genre for this
book")
    # ManyToManyField used because genre can contain many books. Books can
cover many genres.
    # Genre class has already been defined so we can specify the object
above.

    def __str__(self):
        """
        String for representing the Model object.
        """
        return self.title

    def get_absolute_url(self):
        """
        Returns the url to access a particular book instance.
        """
        return reverse('book-detail', args=[str(self.id)])
```

The genre is a `ManyToManyField`, so that a book can have multiple genres and a genre can have many books. The author is declared as `ForeignKey`, so each book will only have only one author, but an author may have many books (in practice a book might have multiple authors, but not in this implementation!)

In both field types the related model class is declared as the first unnamed parameter using either the model class or a string containing the name of the related model. You must use the name of the model as a string if the associated class has not yet been defined in this file before it is referenced! The other parameters of interest in the `author` field are `null=True`, which allows the

database to store a `Null` value if no author is selected, and `on_delete=models.SET_NULL`, which will set the value of the author to `Null` if the associated author record is deleted.

The model also defines `__str__()`, using the book's `title` field to represent a `Book` record. The final method, `get_absolute_url()` returns a URL that can be used to access a detail record for this model (for this to work we will have to define a URL mapping that has the name `book-detail`, and define an associated view and template).

BookInstance model

Next, copy the `BookInstance` model (shown below) under the other models. The `BookInstance` represents a specific copy of a book that someone might borrow, and includes information about whether the copy is available or on what date it is expected back, "imprint" or version details, and a unique id for the book in the library.

Some of the fields and methods will now be familiar. The model uses

- `ForeignKey` to identify the associated `Book` (each book can have many copies, but a copy can only have one `Book`).
- `CharField` to represent the imprint (specific release) of the book.

```
import uuid # Required for unique book instances

class BookInstance(models.Model):
    """
    Model representing a specific copy of a book (i.e. that can be borrowed
    from the library).
    """
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
        help_text="Unique ID for this particular book across whole library")
    book = models.ForeignKey('Book', on_delete=models.SET_NULL, null=True)
    imprint = models.CharField(max_length=200)
    due_back = models.DateField(null=True, blank=True)

    LOAN_STATUS = (
        ('d', 'Maintenance'),
        ('o', 'On loan'),
        ('a', 'Available'),
        ('r', 'Reserved'),
    )

    status = models.CharField(max_length=1, choices=LOAN_STATUS, blank=True,
        default='d', help_text='Book availability')

    class Meta:
        ordering = ["due_back"]

    def __str__(self):
        """
        String for representing the Model object
        """
        return '%s (%s)' % (self.id, self.book.title)
```

We additionally declare a few new types of field:

- `UUIDField` is used for the `id` field to set it as the `primary_key` for this model. This type of field allocates a globally unique value for each instance (one for every book you can find in the library).
- `DateField` is used for the `due_back` date (at which the book is expected to come available after being borrowed or in maintenance). This value can be `blank` or `null` (needed for when the book is available). The model metadata (`Class Meta`) uses this field to order records when they are returned in a query.
- `status` is a `CharField` that defines a choice/selection list. As you can see, we define a tuple containing tuples of key-value pairs and pass it to the `choices` argument. The value in a key/value pair is a display value that a user can select, while the keys are the values that are actually saved if the option is selected. We've also set a default value of 'd' (maintenance) as books will initially be created unavailable before they are stocked on the shelves.

The model `__str__()` represents the `BookInstance` object using a combination of its unique id and the associated `Book`'s title.

Note: A little Python:

- The value returned by `__str__()` is a *formatted string*. Within the string we use `%s` to declare "placeholders". After the string we specify `%` and then a tuple containing the values to be inserted in the placeholders. If you just have one placeholder then you can omit the tuple — e.g. `'My value: %s' % variable`.

Note also that although this approach is perfectly valid, please be aware that it is no longer preferred. Since Python 3 you should instead use the `format` method, eg. `'{1} {2}'.format(self.id, self.book.title)`. You can read more about it [here](#).

Author model

Copy the `Author` model (shown below) underneath the existing code in **`models.py`**.

All of the fields/methods should now be familiar. The model defines an author as having a first name, last name, date of birth, and (optional) date of death. It specifies that by default the `__str__()` returns the name in *last name, firstname* order. The `get_absolute_url()` method reverses the author-detail URL mapping to get the URL for displaying an individual author.

```
class Author(models.Model):
    """
    Model representing an author.
    """
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True, blank=True)
    date_of_death = models.DateField('Died', null=True, blank=True)

    def get_absolute_url(self):
        """
        Returns the url to access a particular author instance.
        """
        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        """
```



```
String for representing the Model object.
"""
return '%s, %s' % (self.last_name, self.first_name)
```

[Re-run the database migrations](#)[Edit](#)

All your models have now been created. Now re-run your database migrations to add them to your database.

```
python3 manage.py makemigrations
python3 manage.py migrate
```

[Language model — challenge](#)[Edit](#)

Imagine a local benefactor donates a number of new books written in another language (say, Farsi). The challenge is to work out how these would be best represented in our library website, and then to add them to the models.

Some things to consider:

- Should "language" be associated with a `Book`, `BookInstance`, or some other object?
- Should the different languages be represented using model, a free text field, or a hard-coded selection list?

After you've decided, add the field. You can see what we decided on Github [here](#).

[Summary](#)[Edit](#)

In this article we've learned how models are defined, and then used this information to design and implement appropriate models for the *LocalLibrary* website.

At this point we'll divert briefly from creating the site, and check out the *Django Administration site*. This site will allow us to add some data to the library, which we can then display using our (yet to be created) views and templates.

Django Tutorial Part 4: Django admin site

Now that we've created models for the [LocalLibrary](#) website, we'll use the Django Admin site to add some "real" book data. First we'll show you how to register the models with the admin site, then we'll show you how to login and create some data. At the end of the article we will show some of ways you can further improve the presentation of the Admin site.

Prerequisites: First complete: [Django Tutorial Part 3: Using models](#).

Objective: To understand the benefits and limitations of the Django admin site, and use it to create some records for our models.

Overview [Edit](#)

The Django admin *application* can use your models to automatically build a site area that you can use to create, view, update, and delete records. This can save you a lot of time during development, making it very easy to test your models and get a feel for whether you have the *right* data. The admin application can also be useful for managing data in production, depending on the type of website. The Django project recommends it only for internal data management (i.e. just for use by admins, or people internal to your organization), as the model-centric approach is not necessarily the best possible interface for all users, and exposes a lot of unnecessary detail about the models.

All the configuration required to include the admin application in your website was done automatically when you [created the skeleton project](#) (for information about actual dependencies needed, see the [Django docs here](#)). As a result, all you **must** do to add your models to the admin application is to *register* them. At the end of this article we'll provide a brief demonstration of how you might further configure the admin area to better display our model data.

After registering the models we'll show how to create a new "superuser", login to the site, and create some books, authors, book instances, and genres. These will be useful for testing the views and templates we'll start creating in the next tutorial.

Registering models [Edit](#)

First, open **admin.py** in the catalog application (`/locallibrary/catalog/admin.py`). It currently looks like this — note that it already imports `django.contrib.admin`:

```
from django.contrib import admin

# Register your models here.
```

Register the models by copying the following text into the bottom of the file. This code simply imports the models and then calls `admin.site.register` to register each of them.

```
from .models import Author, Genre, Book, BookInstance
```

```
admin.site.register(Book)
admin.site.register(Author)
admin.site.register(Genre)
admin.site.register(BookInstance)
```

Note: If you accepted the challenge to create a model to represent the natural language of a book ([see the models tutorial article](#)), import and register it too!

This is the simplest way of registering a model, or models, with the site. The admin site is highly customisable, and we'll talk more about the other ways of registering your models further down.

Creating a superuser [Edit](#)

In order to log into the admin site, we need a user account with *Staff* status enabled. In order to view and create records we also need this user to have permissions to manage all our objects. You can create a "superuser" account that has full access to the site and all needed permissions using **manage.py**.

Call the following command, in the same directory as **manage.py**, to create the superuser. You will be prompted to enter a username, email address, and *strong* password.

```
python3 manage.py createsuperuser
```

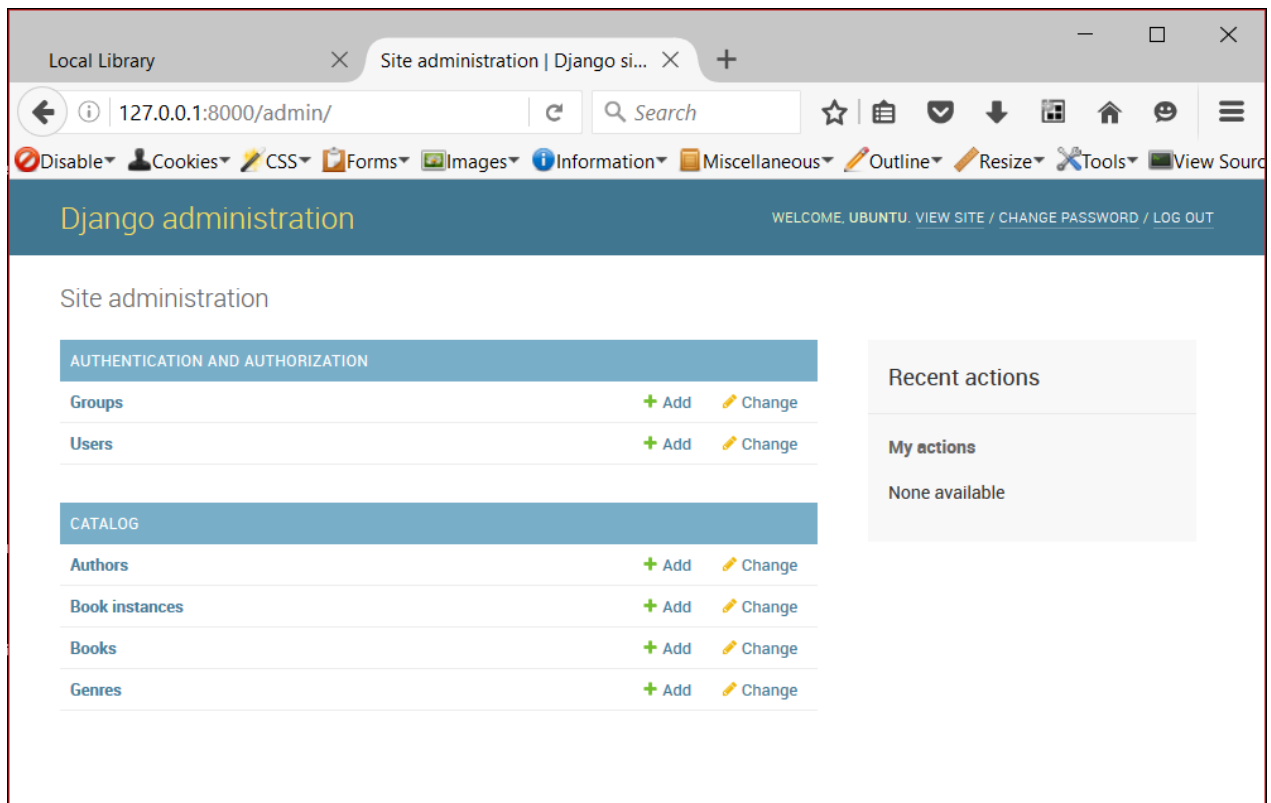
Once this command completes a new superuser will have been added to the database. Now restart the development server so we can test the login:

```
python3 manage.py runserver
```

Logging in and using the site [Edit](#)

To login to the site, open the */admin* URL (e.g. <http://127.0.0.1:8000/admin>) and enter your new superuser userid and password credentials (you'll be redirected to the *login* page, and then back to the */admin* URL after you've entered your details).






This part of the site displays all our models, grouped by installed application. You can click on a model name to go to a screen that lists all its associated records, and you can further click on those records to edit them. You can also directly click the **Add** link next to each model to start creating a record of that type.



Click on the **Add** link to the right of *Books* to create a new book (this will display a dialog much like the one below). Note how the titles of each field, the type of widget used, and the `help_text` (if any) match the values you specified in the model.

Enter values for the fields. You can create new authors or genres by pressing the + button next to the respective fields (or select existing values from the lists if you've already created them). When you're done you can press **SAVE**, **Save and add another**, or **Save and continue editing** to save the record.

Add book

Title:	<input type="text"/>
Author:	<input type="text" value="-----"/>    
Summary:	<div><div></div><div>Enter a brief description of the book</div></div>
ISBN:	<input type="text"/> <small>13 Character ISBN number</small>
Genre:	<div><div>Science Fiction Fantasy Western French Poetry</div><div></div></div> <small>Select a genre for this book. Hold down "Control", or "Command" on a Mac, to select more than one.</small>
<div><div>Save and add another</div><div>Save and continue editing</div><div>SAVE</div></div>	

Note: At this point we'd like you to spend some time adding a few books, authors, and genres (e.g. Fantasy) to your application. Make sure that each author and genre includes a couple of different books (this will make your list and detail views more interesting when we implement them later on in the article series).

When you've finished adding books, click on the **Home** link in the top bookmark to be taken back to the main admin page. Then click on the **Books** link to display the current list of books (or on one of the other links to see other model lists). Now that you've added a few books, the list might look similar to the screenshot below. The title of each book is displayed; this is the value returned in the Book model's `__str__()` method that we specified in the last article.

Select book to change

ADD BOOK +

Action: 0 of 3 selected

<input type="checkbox"/>	BOOK
<input type="checkbox"/>	The Wise Man's Fear
<input type="checkbox"/>	The Dueling Machine
<input type="checkbox"/>	The Name of the Wind
3 books	

From this list you can delete books by selecting the checkbox next to the book you don't want, selecting the *delete...* action from the *Action* drop-down list, and then pressing the **Go** button. You can also add new books by pressing the **ADD BOOK** button.

You can edit a book by selecting its name in the link. The edit page for a book, shown below, is almost identical to the "Add" page. The main differences are the page title (*Change book*) and the addition of **Delete**, **HISTORY** and **VIEW ON SITE** buttons (this last button appears because we defined the `get_absolute_url()` method in our model).

Change book

HISTORY

VIEW ON SITE >

Title:

The Name of the Wind

Author:

Rothfuss, Patrick



Summary:

Told in Kvothe's own voice, this is the tale of the magically gifted young man who grows to be the most notorious wizard his world has ever seen.

Enter a brief description of the book

ISBN:

9781473211896

13 Character ISBN number

Genre:

Science Fiction
Fantasy
Western
French Poetry



Select a genre for this book Hold down "Control", or "Command" on a Mac, to select more than one.

Delete

Save and add another







Save and continue editing

SAVE

Now navigate back to the **Home** page (using the *Home* link the breadcrumb trail) and then view the **Author** and **Genre** lists — you should already have quite a few created from when you added the new books, but feel free to add some more.

What you won't have is any *Book Instances*, because these are not created from Books (although you can create a `Book` from a `BookInstance` — this is the nature of the `ForeignKey` field). Navigate back to the *Home* page and press the associated **Add** button to display the *Add book instance* screen below. Note the large, globally unique Id, which can be used to separately identify a single copy of a book in the library.

Add book instance

Id:	<input type="text" value="945bb5fe804949808c276aa"/>
	<small>Unique ID for this particular book across whole library</small>
Book:	<div><input type="text" value="-----"/>    </div>
Imprint:	<input type="text"/>
Due back:	<input type="text"/> <small>Today</small> 
	<small>Note: You are 10 hours ahead of server time.</small>
Status:	<div><input type="text" value="Maintenance"/> </div> <small>Book availability</small>

Create a number of these records for each of your books. Set the status as *Available* for at least some records and *On loan* for others. If the status is **not** *Available*, then also set a future *Due back* date.

That's it! You've now learned how to set up and use the administration site. You've also created records for `Book`, `BookInstance`, `Genre`, and `Author` that we'll be able to use once we create our own views and templates.

Advanced configuration [Edit](#)

Django does a pretty good job of creating a basic admin site using the information from the registered models:

- Each model has a list of individual records, identified by the string created with the model's `__str__()` method, and linked to detail views/forms for editing. By default, this view has an action menu up the top that you can use to perform bulk delete operations on records.
- The model detail record forms for editing and adding records contain all the fields in the model, laid out vertically in their declaration order.

You can further customise the interface to make it even easier to use. Some of the things you can do are:

- List views:
 - Add additional fields/information displayed for each record.
 - Add filters to select which records are listed, based on date or some other selection value (e.g. Book loan status).

- Add additional options to the actions menu in list views and choose where this menu is displayed on the form.
- Detail views
 - Choose which fields to display (or exclude), along with their order, grouping, whether they are editable, the widget used, orientation etc.
 - Add related fields to a record to allow inline editing (e.g. add the ability to add and edit book records while you're creating their author record).

In this section we're going to look at a few changes that will improve the interface for our *LocalLibrary*, including adding more information to `Book` and `Author` model lists, and improving the layout of their edit views. We won't change the `Language` and `Genre` model presentation because they only have one field each, so there is no real benefit in doing so!

You can find a complete reference of all the admin site customisation choices in [The Django Admin site](#) (Django Docs).

Register a `ModelAdmin` class

To change how a model is displayed in the admin interface you define a [ModelAdmin](#) class (which describes the layout) and register it with the model.

Let's start with the `Author` model. Open **admin.py** in the catalog application (`/locallibrary/catalog/admin.py`). Comment out your original registration (prefix it with a `#`) for the `Author` model:

```
# admin.site.register(Author)
```

Now add a new `AuthorAdmin` and registration as shown below.

```
# Define the admin class
class AuthorAdmin(admin.ModelAdmin):
    pass

# Register the admin class with the associated model
admin.site.register(Author, AuthorAdmin)
```

Now we'll add `ModelAdmin` classes for `Book`, and `BookInstance`. We again need to comment out the original registrations:

```
#admin.site.register(Book)
#admin.site.register(BookInstance)
```

Now to create and register the new models; for the purpose of this demonstration, we'll instead use the `@register` decorator to register the models (this does exactly the same thing as the `admin.site.register()` syntax):

```
# Register the Admin classes for Book using the decorator
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    pass

# Register the Admin classes for BookInstance using the decorator
```

```
@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    pass
```

Currently all of our admin classes are empty (see "pass") so the admin behaviour will be unchanged! We can now extend these to define our model-specific admin behaviour.

Configure list views

The *LocalLibrary* currently lists all authors using the object name generated from the model `__str__()` method. This is fine when you only have a few authors, but once you have many you may end up having duplicates. To differentiate them, or just because you want to show more interesting information about each author, you can use [list_display](#) to add additional fields to the view.

Replace your `AuthorAdmin` class with the code below. The field names to be displayed in the list are declared in a tuple in the required order, as shown (these are the same names as specified in your original model).

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('last_name', 'first_name', 'date_of_birth',
                   'date_of_death')
```

Relaunch the site and navigate to the author list. The fields above should now be displayed, like so:

[Home](#) › [Catalog](#) › [Authors](#)

Select author to change

ADD AUTHOR +

Action:

Go

0 of 2 selected

<input type="checkbox"/>	LAST NAME	FIRST NAME	DATE OF BIRTH	DIED
<input type="checkbox"/>	Bova	Ben	Nov. 8, 2032	-
<input type="checkbox"/>	Rothfuss	Patrick	June 6, 2073	-

2 authors

For our `Book` model we'll additionally display the `author` and `genre`. The `author` is a `ForeignKey` field (one-to-one) relationship, and so will be represented by the `__str__()` value for the associated record. Replace the `BookAdmin` class with the version below.

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'display_genre')
```

Unfortunately we can't directly specify the `genre` field in `list_display` because it is a `ManyToManyField` (Django prevents this because there would be a large database access "cost" in doing so). Instead we'll define a `display_genre` function to get the information as a string (this is the function we've called above; we'll define it below).

Note: Getting the `genre` may not be a good idea here, because of the "cost" of the database operation. We're showing you how because calling functions in your models can be very useful for other reasons — for example to add a *Delete* link next to every item in the list.

Add the following code into your `Book` model (**models.py**). This creates a string from the first three values of the `genre` field (if they exist) and creates a `short_description` that can be used in the admin site for this method.

```
def display_genre(self):
    """
    Creates a string for the Genre. This is required to display genre in
    Admin.
    """
    return ', '.join([ genre.name for genre in self.genre.all()[:3] ])
    display_genre.short_description = 'Genre'
```

After saving the model and updated admin, relaunch the site and go to the *Books* list page; you should see a book list like the one below:

Home » Catalog » Books

Select book to change ADD BOOK +

Action: 0 of 3 selected

<input type="checkbox"/>	TITLE	AUTHOR	GENRE
<input type="checkbox"/>	The Dueling Machine	Bova, Ben	Science Fiction
<input type="checkbox"/>	The Wise Man's Fear (The Kingkiller Chronicle, #2)	Rothfuss, Patrick	Fantasy
<input type="checkbox"/>	The Name of the Wind (The Kingkiller Chronicle, #1)	Rothfuss, Patrick	Fantasy

3 books

The `Genre` model (and the `Language` model, if you defined one) both have a single field, so there is no point creating an additional model for them to display additional fields.

Note: It is worth updating the `BookInstance` model list to show at least the status and the expected return date. We've added that as a challenge at the end of this article!

Add list filters

Once you've got a lot of items in a list, it can be useful to be able to filter which items are displayed. This is done by listing fields in the `list_filter` attribute. Replace your current `BookInstanceAdmin` class with the code fragment below.

```
class BookInstanceAdmin(admin.ModelAdmin):
    list_filter = ('status', 'due_back')
```

The list view will now include a filter box to the right. Note how you can choose dates and status to filter the values:

Select book instance to change

ADD BOOK INSTANCE +

Action: 0 of 2 selected

<input type="checkbox"/>	BOOK	STATUS	DUE BACK	ID
<input type="checkbox"/>	The Wise Man's Fear (The Kingkiller Chronicle, #2)	Maintenance	Oct. 2, 2016	e5d22dd1c8f5-4e9aed1-f77240ec
<input type="checkbox"/>	The Name of the Wind (The Kingkiller Chronicle, #1)	Maintenance	Oct. 12, 2016	99841f53c533-494b33d-687946f1

2 book instances

FILTER

By status

All
 Maintenance
 On loan
 Available
 Reserved

By due back

Any date
 Today
 Past 7 days
 This month
 This year
 No date
 Has date

Organise detail view layout

By default, the detail views lay out all fields vertically, in their order of declaration in the model. You can change the order of declaration, which fields are displayed (or excluded), whether sections are used to organise the information, whether fields are displayed horizontally or vertically, and even what edit widgets are used in the admin forms.

Note: The *LocalLibrary* models are relatively simple so there isn't a huge need for us to change the layout; we'll make some changes anyway however, just to show you how.

Controlling which fields are displayed and laid out

Update your `AuthorAdmin` class to add the `fields` line, as shown below (in bold):

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('last_name', 'first_name', 'date_of_birth',
                  'date_of_death')
    fields = ('first_name', 'last_name', ('date_of_birth', 'date_of_death'))
```

The `fields` attribute lists just those fields that are to be displayed on the form, in order. Fields are displayed vertically by default, but will display horizontally if you further group them in a tuple (as shown in the "date" fields above).

Restart your application and go to the author detail view — it should now appear as shown below:

Change author

HISTORY

VIEW ON SITE >

First name: Last name: Date of birth: Today | Died: Today | 

Note: You are 11 hours ahead of server time.

Note: You are 11 hours ahead of server time.

Note: You can also use the `exclude` attribute to declare a list of attributes to be excluded from the form (all other attributes in the model will be displayed).

Sectioning the detail view

You can add "sections" to group related model information within the detail form, using the [fieldsets](#) attribute.

In the `BookInstance` model we have information related to what the book is (i.e. `name`, `imprint`, and `id`) and when it will be available (`status`, `due_back`). We can add these in different sections by adding the text in bold to our `BookInstanceAdmin` class.

```
@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    list_filter = ('status', 'due_back')




    fieldsets = (
        (None, {
            'fields': ('book', 'imprint', 'id')
        }),
        ('Availability', {
            'fields': ('status', 'due_back')
        }),
    )
```

Each section has its own title (or `None`, if you don't want a title) and an associated tuple of fields in a dictionary — the format is complicated to describe, but fairly easy to understand if you look at the code fragment immediately above.


Restart and navigate to a book instance view; the form should appear as shown below:

Change book instance

HISTORY

Book:   Imprint: Id:
Unique ID for this particular book across whole library

Availability

Status: 
Book availabilityDue back: 
Note: You are 11 hours ahead of server time.

Delete

Save and add another

Save and continue editing

SAVE

Inline editing of associated records

Sometimes it can make sense to be able to add associated records at the same time. For example, it may make sense to have both the book information and information about the specific copies you've got on the same detail page.

You can do this by declaring [inlines](#), of type [TabularInline](#) (horizontal layout) or [StackedInline](#) (vertical layout, just like the default model layout). You can add the `BookInstance` information inline to our `Book` detail by adding the lines below in bold near your `BookAdmin`:

```
class BooksInstanceInline(admin.TabularInline):
    model = BookInstance

@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'display_genre')
    inlines = [BooksInstanceInline]
```

Try relaunching your app then looking at the view for a `Book` — at the bottom you should now see the book instances relating to this book:

Genre:

Fantasy
Science Fiction

+

Select a genre for this book Hold down "Control", or "Command" on a Mac, to select more than one.

BOOK INSTANCES

ID	IMPRINT	DUE BACK	STATUS	DELETE?
344edc33-37a0-497f-b574-637b6ddcf285 (The Name of the Wind (The Kingkiller Chronicle, #1))	Published March 27th 2007 by Penguin Grou	Today I	Available	<input type="checkbox"/>
Note: You are 11 hours ahead of server time.				
99841f53-c533-494b-b33d-687946f19bed (The Name of the Wind (The Kingkiller Chronicle, #1))	Published March 27th 2007 by Penguin Grou	2016-10-12 Today I	Maintenance	<input type="checkbox"/>
Note: You are 11 hours ahead of server time.				
8edafcc6-9d1e-4056-9f20-ce99d1ef157e (The Name of the Wind (The Kingkiller Chronicle, #1))	Published March 27th 2007 by Penguin Grou	2016-10-12 Today I	On loan	<input type="checkbox"/>
Note: You are 11 hours ahead of server time.				
2d1adecdc28fd48d2bd43ae		Today I	Maintenance	
Note: You are 11 hours ahead of server time.				
63f8b2a43e4f497d98b001		Today I	Maintenance	
Note: You are 11 hours ahead of server time.				
2e8cb69ce978416a86dfef		Today I	Maintenance	
Note: You are 11 hours ahead of server time.				

+ Add another Book instance

In this case all we've done is declare our tabular inline class, which just adds all fields from the *inlined* model. You can specify all sorts of additional information for the layout, including the fields to display, their order, whether they are read only or not, etc. (see [TabularInline](#) for more information).

Note: There are some painful limits in this functionality! In the screenshot above we have three existing book instances, followed by three placeholders for new book instances (which look very similar!). It would be better to have NO spare book instances by default and just add them with the **Add another Book instance** link, or to be able to just list the `BookInstances` as non-readable links from here. The first option can be done by setting the `extra` attribute to 0 in `BookInstanceInline` model, try it by yourself.

Challenge yourself [Edit](#)

We've learned a lot in this section, so now it is time for you to try a few things.

1. For the `BookInstance` list view, add code to display the book, status, due back date, and id (rather than the default `__str__()` text).
2. Add an inline listing of `Book` items to the `Author` detail view using the same approach as we did for `Book/BookInstance`.

[Summary](#)[Edit](#)

That's it! You've now learned how to set up the administration site in both its simplest and improved form, how to create a superuser, and how to navigate the admin site and view, delete, and update records. Along the way you've created a bunch of Books, BookInstances, Genres and Authors that we'll be able to list and display once we create our own view and templates.

Django Tutorial Part 5: Creating our home page

We're now ready to add the code to display our first full page — a home page for the [LocalLibrary](#) website that shows how many records we have of each model type and provides sidebar navigation links to our other pages. Along the way we'll gain practical experience in writing basic URL maps and views, getting records from the database, and using templates.

Prerequisites: Read the [Django Introduction](#). Complete previous tutorial topics (including [Django Tutorial Part 4: Django admin site](#)).

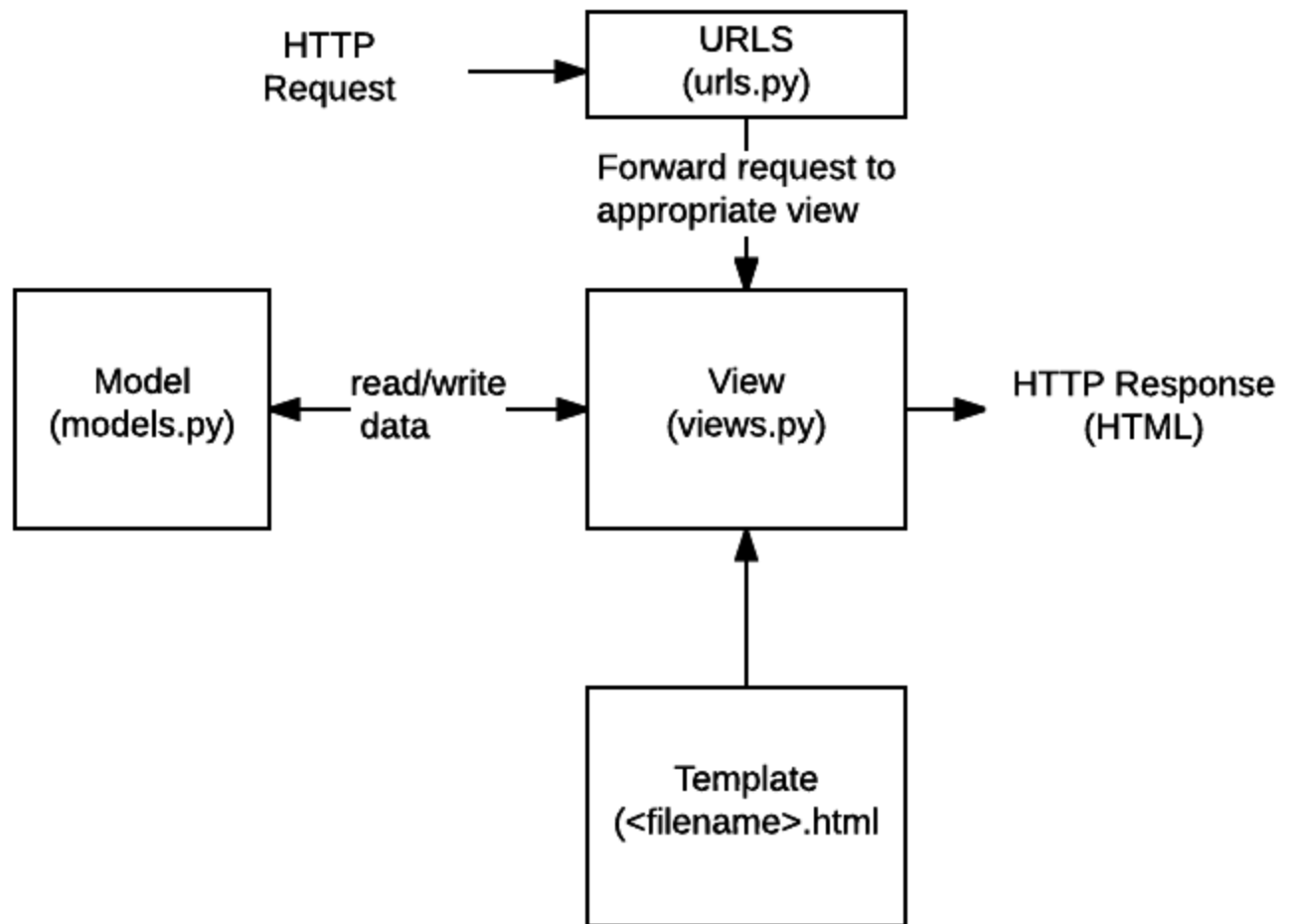
Objective: To understand how to create simple url maps and views (where no data is encoded in the URL), and how to get data from models and create templates.

[Overview](#)[Edit](#)

Now we have defined our models and created some initial library records to work with, it's time to write the code to present that information to users. The first thing we need to do is determine what information we want to be able to display in our pages, and then define appropriate URLs for returning those resources. Then we're going to need to create the url mapper, views, and templates to display those pages.

The diagram below is provided as a reminder of the main flow of data and things that need to be implemented when handling an HTTP request/response. As we've already created the model, the main things we'll need to create are:

- URL maps to forward the supported URLs (and any information encoded in the URLs) to the appropriate view functions.
- View functions to get the requested data from the models, create an HTML page displaying the data, and return it to the user to view in the browser.
- Templates used by the views to render the data.



As you'll see in the next section, we're going to have 5 pages to display, which is a lot to document in one article. Therefore, most of this article will concentrate on showing you how to implement just the home page (we'll move onto the other pages in a subsequent article). This should give you a good end-to-end understanding of how URL mappers, views, and models work in practice.

Defining the resource URLs [Edit](#)

As this version of *LocalLibrary* is essentially read-only for end users, we just need to provide a landing page for the site (a home page), and pages that *display* list and detail views for books and authors.

The URLs that we're going to need for our pages are:

- `catalog/` — The home/index page.
- `catalog/books/` — The list of all books.
- `catalog/authors/` — The list of all authors.
- `catalog/book/<id>` — The detail view for the specific book with a field primary key of `<id>` (the default). So for example, `/catalog/book/3`, for the third book added.
- `catalog/author/<id>` — The detail view for the specific author with a primary key field named `<id>`. So for example, `/catalog/author/11`, for the 11th author added.

The first three URLs are used to list the index, books, and authors. These don't encode any additional information, and while the results returned will depend on the content in the database, the queries run to get the information will always be the same.

By contrast the final two URLs are used to display detailed information about a specific book or author — these encode the identity of the item to display in the URL (shown as `<id>` above). The URL mapper can extract the encoded information and pass it to the view, which will then dynamically determine what information to get from the database. By encoding the information in our URL we only need one url mapping, view, and template to handle every book (or author).

Note: Django allows you to construct your URLs any way you like — you can encode information in the body of the URL as shown above or use URL `GET` parameters (e.g. `/book/?id=6`). Whichever approach you use, the URLs should be kept clean, logical and readable ([check out the W3C advice here](#)).

The Django documentation tends to recommend encoding information in the body of the URL, a practice that they feel encourages better URL design.

As discussed in the overview, the rest of this article describes how we construct the index page.

Creating the index page [Edit](#)

The first page we'll create will be the index page (`catalog/`). This will display a little static HTML, along with some calculated "counts" of different records in the database. To make this work we'll have to create an URL mapping, view and template.

Note: It's worth paying a little extra attention in this section. Some of the material is common to all of the pages.

URL mapping

We created a basic `/catalog/urls.py` file for our catalog application when we created [the skeleton website](#). The catalog application URLs were included into the project with a mapping of `catalog/`, so the URLs that get to this mapper must have started with `catalog/` (the mapper is working on all strings in the URL after the forward slash).

Open **`urls.py`** and paste in the line shown in bold below.

```
urlpatterns = [  
    url(r'^$', views.index, name='index'),  
]
```

This `url()` function defines a URL pattern (`r'^$',`), and a view function that will be called if the pattern is detected (`views.index` — a function named `index()` in **`views.py`**). The URL pattern is a [Python regular expression](#) (RE). We'll talk a bit more about REs further along in this tutorial, but for this case all you need to know is that an RE of `^$` will pattern match against an empty string (`^` is a string start marker and `$` is an end of string marker).

Note: The matched URL is actually `catalog/ + <empty string>` (because we are in the **catalog** application, `/catalog/` is assumed). Our first view function will be called if we receive an HTTP request with a URL of `/catalog/`.

This `url()` function also specifies a `name` parameter, which uniquely identifies *this* particular URL mapping. You can use this name to "reverse" the mapper — to dynamically create a URL pointing to the resource the mapper is designed to handle. For example, with this in place we can now link to our home page by creating the following link in our template:

```
<a href="{% url 'index' %}">Home</a>.
```

Note: We could of course hard code the above link (e.g. `Home`), but then if we changed the pattern for our home page (e.g. to `/catalog/index`) the templates would no longer link correctly. Using a reversed url mapping is much more flexible and robust!

View (function-based)

A view is a function that processes an HTTP request, fetches data from the database as needed, generates an HTML page by rendering this data using an HTML template, and then returns the HTML in an HTTP response to be shown to the user. The index view follows this model — it fetches information about how many `Book`, `BookInstance`, available `BookInstance` and `Author` records we have in the database, and passes them to a template for display.

Open **catalog/views.py**, and note that the file already imports the [render\(\)](#) shortcut function which generates HTML files using a template and data.

```
from django.shortcuts import render

# Create your views here.
```

Copy the following code at the bottom of the file. The first line imports the model classes that we will use to access data in all our views.

```
from .models import Book, Author, BookInstance, Genre

def index(request):
    """
    View function for home page of site.
    """
    # Generate counts of some of the main objects
    num_books=Book.objects.all().count()
    num_instances=BookInstance.objects.all().count()
    # Available books (status = 'a')
    num_instances_available=BookInstance.objects.filter(status__exact='a').count()
    num_authors=Author.objects.count() # The 'all()' is implied by default.

    # Render the HTML template index.html with the data in the context
    variable = {
        'num_books': num_books,
        'num_instances': num_instances,
        'num_instances_available': num_instances_available,
        'num_authors': num_authors
    }
    return render(
        request,
        'index.html',
        variable)
```

```
context={'num_books':num_books,'num_instances':num_instances,'num_instances_a
vailable':num_instances_available,'num_authors':num_authors},
)
```

The first part of the view function fetches counts of records using the `objects.all()` attribute on the model classes. It also gets a list of `BookInstance` objects that have a status field value of 'a' (Available). You can find out a little bit more about how to access from models in our previous tutorial ([Django Tutorial Part 3: Using models > Searching for records](#)).

At the end of the function we call the `render()` function to create and return an HTML page as a response (this shortcut function wraps a number of other functions, simplifying this very common use-case). This takes as parameters the original `request` object (an `HttpRequest`), an HTML template with placeholders for the data, and a `context` variable (a Python dictionary containing the data that will be inserted into those placeholders).

We'll talk more about templates and the context variable in the next section; let's create our template so we can actually display something to the user!

Template

A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. Django will automatically look for templates in a directory named '**templates**' in your application. So for example, in the index view we just added, the `render()` function will expect to be able to find the file **`/locallibrary/catalog/templates/index.html`**, and will raise an error if the file cannot be found. You can see this if you save the previous changes and go back to your browser — accessing `127.0.0.1:8000` will now give you a fairly intuitive error message "TemplateDoesNotExist at /catalog/", plus other details.

Note: Django will look in a number of places for templates, based on your project's settings file (searching in your installed applications is a default setting!). You can find out more about how Django finds templates and what template formats it supports in [Templates](#) (Django docs).

Extending templates

The index template is going to need standard HTML markup for the head and body, along with sections for navigation (to the other pages in the site that we haven't yet created) and for displaying some introductory text and our book data. Much of this text (the HTML and navigation structure) will be the same for every page on our site. Rather than forcing developers to duplicate this "boilerplate" in every page, the Django templating language allows you to declare a base template, and then extend it, replacing just the bits that are different for each specific page.

For example, a base template **`base_generic.html`** might look like the text below. As you can see, this contains some "common" HTML and sections for title, sidebar and content marked up using named `block` and `endblock` template tags (shown in bold). The blocks can be empty, or contain content that will be used "by default" for derived pages.

Note: Template *tags* are like functions that you can use in a template to loop through lists, perform conditional operations based on the value of a variable, etc. In addition to template tags the template syntax allows you to reference template variables (that are passed into the template from the view) and use *template filters*, which reformat variables (for example, setting a string to lower case).

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block title %}<title>Local Library</title>{% endblock %}
</head>

<body>
    {% block sidebar %}<!-- insert default navigation text for every page -->{%
endblock %}
    {% block content %}<!-- default content text (typically empty) -->{%
endblock %}
</body>
</html>
```

When we want to define a template for a particular view, we first specify the base template (with the `extends` template tag — see the next code listing). If there are any sections that we want to replace in the template we declare these, using identical `block/endblock` sections as used in the base template.

For example, the code fragment below shows how we use the `extends` template tag, and override the `content` block. The final HTML produced would have all the HTML and structure defined in the base template (including the default content you've defined inside the `title` block), but with your new `content` block inserted in place of the default one.

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

{% block content %}
<h1>Local Library Home</h1>
<p>Welcome to <em>LocalLibrary</em>, a very basic Django website developed as
a tutorial example on the Mozilla Developer Network.</p>
{% endblock %}
```

The LocalLibrary base template

The base template we plan to use for the *LocalLibrary* website is listed below. As you can see, this contains some HTML and defined blocks for `title`, `sidebar`, and `content`. We have a default title (which we may want to change) and a default sidebar with links to lists of all books and authors (which we will probably not want to change, but we've allowed scope to do so if needed by putting this in a block).

Note: We also introduce two additional template tags: `url` and `load static`. These are discussed in following sections.

Create a new file — */locallibrary/catalog/templates/base_generic.html* — and give it the following contents:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```

    {% block title %}<title>Local Library</title>{% endblock %}
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></scr
ipt>
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></s
cript>

    <!-- Add additional CSS in static file -->
    {% load static %}
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
</head>

<body>

    <div class="container-fluid">

        <div class="row">
            <div class="col-sm-2">
                {% block sidebar %}
                <ul class="sidebar-nav">
                    <li><a href="{% url 'index' %}">Home</a></li>
                    <li><a href="">All books</a></li>
                    <li><a href="">All authors</a></li>
                </ul>
                {% endblock %}
            </div>
            <div class="col-sm-10">
                {% block content %}{% endblock %}
            </div>
        </div>

    </div>
</body>
</html>

```

The template uses (and includes) JavaScript and CSS from [Bootstrap](#) to improve the layout and presentation of the HTML page. Using Bootstrap or another client-side web framework is a quick way to create an attractive page that can scale well on different browser sizes, and it also allows us to deal with the page presentation without having to get into any of the details — we just want to focus on the server-side code here!

The base template also references a local css file (**styles.css**) that provides a little additional styling. Create **/locallibrary/catalog/static/css/styles.css** and give it the following content:

```

.sidebar-nav {
    margin-top: 20px;
    padding: 0;
    list-style: none;
}

```

The index template

Create the HTML file `/locallibrary/catalog/templates/index.html` and give it the content shown below. As you can see we extend our base template in the first line, and then replace the default content block with a new one for this template.

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

{% block content %}
<h1>Local Library Home</h1>

    <p>Welcome to <em>LocalLibrary</em>, a very basic Django website developed
as a tutorial example on the Mozilla Developer Network.</p>

<h2>Dynamic content</h2>

    <p>The library has the following record counts:</p>
    <ul>
        <li><strong>Books:</strong> {{ num_books }}</li>
        <li><strong>Copies:</strong> {{ num_instances }}</li>
        <li><strong>Copies available:</strong> {{ num_instances_available }}</li>
        <li><strong>Authors:</strong> {{ num_authors }}</li>
    </ul>

{% endblock %}
```

In the *Dynamic content* section we've declared placeholders (*template variables*) for the information we wanted to include from the view. The variables are marked using the "double brace" or "handlebars" syntax (see in bold above).

Note: You can easily recognise whether you're dealing with template variables or template tags (functions) because the variables have double braces (`{{ num_books }}`) while the tags are enclosed in single braces with percentage signs (`{% extends "base_generic.html" %}`).

The important thing to note here is that these variables are named with the *keys* that we passed into the `context` dictionary in our view's `render()` function (see below); these will be replaced by their associated *values* when the template is rendered.

```
return render(
    request,
    'index.html',

    context={
        'num_books': num_books, 'num_instances': num_instances,
        'num_instances_available': num_instances_available,
        'num_authors': num_authors,
    }
)
```

Referencing static files in templates

Your project is likely to use static resources, including JavaScript, CSS, and images. Because the location of these files might not be known (or might change), Django allows you to specify the location of these files in your templates relative to the `STATIC_URL` global setting (the default skeleton website sets the value of `STATIC_URL` to `/static/`, but you might choose to host these on a content delivery network or elsewhere).

Within the template you first call the `load` template tag specifying "static" to add this template library (as shown below). After static is loaded, you can then use the `static` template tag, specifying the relative URL to the file of interest.

```
<!-- Add additional CSS in static file -->
{% load static %}
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

You could if desired add an image into the page in the same sort of fashion. For example:

```
{% load static %}

```

Note: The changes above specify where the files are located, but Django does not serve them by default. While we enabled serving by the development web server in the global URL mapper (`/locallibrary/locallibrary/urls.py`) when we [created the website skeleton](#), you will still need to arrange for them to be served in production. We'll look at this later.

For more information on working with static files see [Managing static files](#) (Django docs).

[Linking to URLs](#)

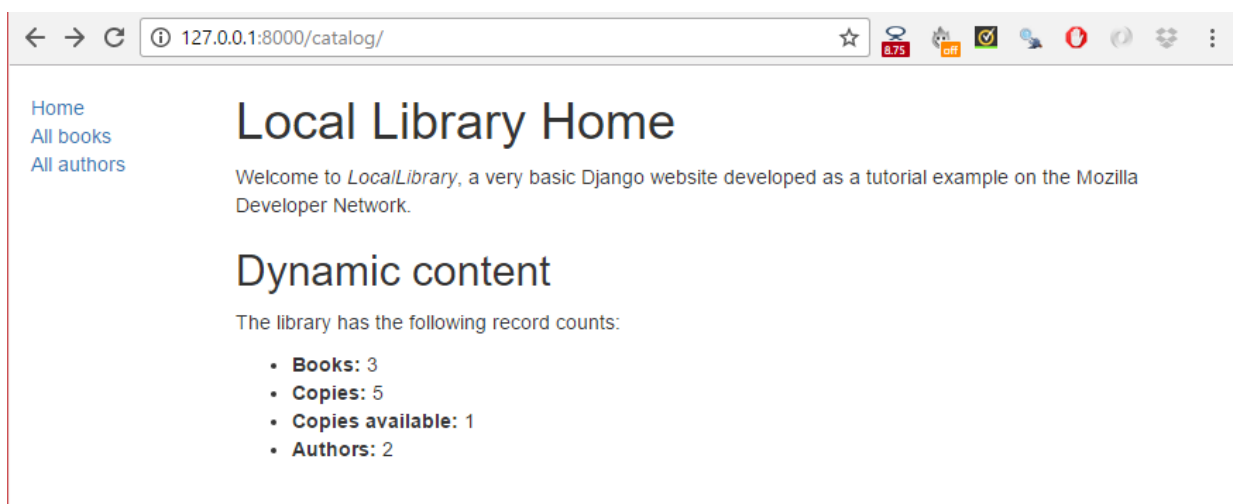
The base template above introduced the `url` template tag.

```
<li><a href="{% url 'index' %}">Home</a></li>
```

This tag takes the name of a `url()` function called in your `urls.py` and values for any arguments the associated view will receive from that function, and returns a URL that you can use to link to the resource.

[What does it look like?](#)[Edit](#)

At this point we should have created everything needed to display the index page. Run the server (`python3 manage.py runserver`) and open your browser to <http://127.0.0.1:8000/>. If everything is set up correctly, your site should look something like the following screenshot.



Note: You won't be able to use the **All books** and **All authors** links yet, because the urls, views, and templates for those pages haven't been defined (currently we've just inserted placeholders for those links in the `base_generic.html` template).

Challenge yourself [Edit](#)

Here are a couple of tasks to test your familiarity with model queries, views and templates.

1. Declare a new title block in the index template and change the page title to match this particular page.
2. Modify the view to generate a count of genres and a count of books that contain a particular word (case insensitive) and then add these fields to the template.

Summary [Edit](#)

We've now created the home page for our site — an HTML page that displays some counts of records from the database and has links to our other still-to-be-created pages. Along the way we've learned a lot of fundamental information about url mappers, views, querying the database using our models, how to pass information to a template from your view, and how to create and extend templates.

Django Tutorial Part 6: Generic list and detail views

This tutorial extends our [LocalLibrary](#) website, adding list and detail pages for books and authors. Here we'll learn about generic class-based views, and show how they can reduce the amount of code you have to write for common use cases. We'll also go into URL handling in greater detail, showing how to perform basic pattern matching.

Prerequisites: Complete all previous tutorial topics, including [Django Tutorial Part 5: Creating our home page](#).

Objective: To understand where and how to use generic class-based views, and how to extract patterns from URLs and pass the information to views.

[Overview](#)[Edit](#)

In this tutorial we're going to complete the first version of the [LocalLibrary](#) website by adding list and detail pages for books and authors (or to be more precise, we'll show you how to implement the book pages, and get you to create the author pages yourself!)

The process is similar to the creating the index page, which we showed in the previous tutorial. We'll still need to create URL maps, views and templates. The main difference is that for the detail pages we'll have the additional challenge of extracting information from patterns in the URL and passing it to the view. For these pages we're going to demonstrate a completely different type of view: generic class-based list and detail views. These can significantly reduce the amount of view code needed, making them easier to write and maintain.

The final part of the tutorial will demonstrate how to paginate your data when using generic class-based list views.

[Book list page](#)[Edit](#)

The book list page will display a list of all the available book records in the page, accessed using the url: `catalog/books/`. The page will display a title and author for each record, with the title being a hyperlink to the associated book detail page. The page will have the same structure and navigation as all other pages in the site, and will hence can extend the base template (**base_generic.html**) we created in the previous tutorial.

[URL mapping](#)

Open `/catalog/urls.py` and copy in the line shown in bold below. Much like our index map, this `url()` function defines a regular expression (RE) to match against the URL (**`r'^books/$'`**), a view function that will be called if the URL matches (`views.BookListView.as_view()`) and a name for this particular mapping.

```
urlpatterns = [  
    url(r'^$', views.index, name='index'),  
    url(r'^books/$', views.BookListView.as_view(), name='books'),  
]
```

]

The regular expression here matches against URLs that equal to `books/` (`^` is a string start marker and `$` is an end of string marker). As discussed in the previous tutorial the URL must already have matched `/catalog`, so the view will actually be called for the URL:
`/catalog/books/`.

The view function has a different format than before — that's because this view will actually be implemented as a class. We will be inheriting from an existing generic view function that already does most of what we want this view function to do, rather than writing our own from scratch.

For Django class-based views we access an appropriate view function by calling the class method `as_view()`. This does all the work of creating an instance of the class, and making sure that the right handler methods are called for incoming HTTP requests.

View (class-based)

We could quite easily write the book list view as a regular function (just like our previous index view), which would query the database for all books, and then call `render()` to pass the list to a specified template. Instead however, we're going to use a class-based generic list view (`ListView`) — a class that inherits from an existing view. Because the generic view already implements most of the functionality we need, and follows Django best-practice, we will be able to create a more robust list view with less code, less repetition, and ultimately less maintenance.

Open **catalog/views.py**, and copy the following code into the bottom of the file:

```
from django.views import generic

class BookListView(generic.ListView):
    model = Book
```

That's it! The generic view will query the database to get all records for the specified model (`Book`) then render a template located at **/locallibrary/catalog/templates/catalog/book_list.html** (which we will create below). Within the template you can access the list of books with the template variable named `object_list` OR `book_list` (i.e. generically "*the_model_name_list*").

Note: This awkward path for the template location isn't a misprint — the generic views look for templates in `/application_name/the_model_name_list.html` (`catalog/book_list.html` in this case) inside the application's `/application_name/templates/` directory (`/catalog/templates/`).

You can add attributes to change the default behaviour above. For example, you can specify another template file if you need to have multiple views that use this same model, or you might want to use a different template variable name if `book_list` is not intuitive for your particular template use-case. Possibly the most useful variation is to change/filter the subset of results that are returned — so instead of listing all books you might list top 5 books that were read by other users.

```
class BookListView(generic.ListView):
    model = Book
```

```
context_object_name = 'my_book_list'    # your own name for the list as a
template variable
queryset = Book.objects.filter(title__icontains='war')[:5] # Get 5 books
containing the title war
template_name = 'books/my_arbitrary_template_name_list.html' # Specify
your own template name/location
Overriding methods in class-based views
```

While we don't need to do so here, you can also override some of the class methods.

For example, we can override the `get_queryset()` method to change the list of records returned. This is more flexible than just setting the `queryset` attribute as we did in the preceding code fragment (though there is no real benefit in this case):

```
class BookListView(generic.ListView):
    model = Book

    def get_queryset(self):
        return Book.objects.filter(title__icontains='war')[:5] # Get 5 books
containing the title war
```

We might also override `get_context_data()` in order to pass additional context variables to the template (e.g. the list of books is passed by default). The fragment below shows how to add a variable named "some_data" to the context (it would then be available as a template variable).

```
class BookListView(generic.ListView):
    model = Book

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(BookListView, self).get_context_data(**kwargs)
        # Get the blog from id and add it to the context
        context['some_data'] = 'This is just some data'
        return context
```

When doing this it is important to follow the pattern used above:

- First get the existing context from our superclass.
- Then add your new context information.
- Then return the new (updated) context.

Note: Check out [Built-in class-based generic views](#) (Django docs) for many more examples of what you can do.

Creating the List View template

Create the HTML file `/locallibrary/catalog/templates/catalog/book_list.html` and copy in the text below. As discussed above, this is the default template file expected by the generic class-based list view (for a model named `Book` in an application named `catalog`).

Templates for generic views are just like any other templates (although of course the context/information passed to the template may differ). As with our *index* template, we extend our base template in the first line, and then replace the block named `content`.

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

{% block content %}
    <h1>Book List</h1>

    {% if book_list %}
    <ul>

        {% for book in book_list %}
        <li>
            <a href="{{ book.get_absolute_url }}">{{ book.title }}</a>
            ({{book.author}})
        </li>
        {% endfor %}

    </ul>
    {% else %}
        <p>There are no books in the library.</p>
    {% endif %}
{% endblock %}
```

The view passes the context (list of books) by default as `object_list` and `book_list` (aliases; either will work).

Conditional execution

We use the [if](#), `else` and `endif` template tags to check whether the `book_list` has been defined and is not empty. It is empty (the `else` clause) then we display text explaining that there are no books to display. If it is not empty, then we iterate through the list of books.

```
{% if book_list %}
    <!-- code here to list the books -->
{% else %}
    <p>There are no books in the library.</p>
{% endif %}
```

The condition above only checks for one case, but you can test on additional conditions using the `elif` template tag (e.g. `{% elif var2 %}`). For more information about conditional operators see: [if](#), [ifequal/ifnotequal](#), and [ifchanged](#) in [Built-in template tags and filters](#) (Django Docs).

For loops

The template uses the [for](#) and `endfor` template tags to loop through the book list, as shown below. Each iteration populates the `book` template variable with information for the current list item.

```
{% for book in book_list %}
    <li> <!-- code here get information from each book item --> </li>
{% endfor %}
```

While not used here, within the loop Django will also create other variables that you can use to track the iteration. For example, you can test the `forloop.last` variable to perform conditional processing the last time that the loop is run.

Accessing variables

The code inside the loop creates a list item for each book that shows both the title (as a link to the yet-to-be-created detail view) and the author.

```
<a href="{{ book.get_absolute_url }}">{{ book.title }}</a> ({{book.author}})
```

We access the *fields* of the associated book record using the "dot notation" (e.g. `book.title` and `book.author`), where the text following the `book` item is the field name (as defined in the model).

We can also call *functions* in the model from within our template — in this case we call `Book.get_absolute_url()` to get an URL you could use to display the associated detail record. This works provided the function does not have any arguments (there is no way to pass arguments!)

Note: We have to be a little careful of "side effects" when calling functions in templates. Here we just get a URL to display, but a function can do pretty much anything — we wouldn't want to delete our database (for example) just by rendering our template!

Update the base template

Open the base template (`/locallibrary/catalog/templates/base_generic.html`) and insert `{% url 'books' %}` into the URL link for **All books**, as shown below. This will enable the link in all pages (we can successfully put this in place now that we've created the "books" url mapper).

```
<li><a href="{% url 'index' %}">Home</a></li>
<li><a href="{% url 'books' %}">All books</a></li>
<li><a href="">All authors</a></li>
```

What does it look like?

You won't be able to build book list yet, because we're still missing a dependency — the URL map for the book detail pages, which is needed to create hyperlinks to individual books. We'll show both list and detail views after the next section.

Book detail page [Edit](#)

The book detail page will display information about a specific book, accessed using the URL `catalog/book/<id>` (where `<id>` is the primary key for the book). In addition to fields in the `Book` model (author, summary, ISBN, language, and genre, we'll also list the details of the available copies (`BookInstances`) including the status, expected return date, imprint, and id. This will allow our readers not just to learn about the book, but to confirm whether/when it is available.

URL mapping

Open `/catalog/urls.py` and add the **'book-detail'** URL mapper shown in bold below. This `url()` function defines a pattern, associated generic class-based detail view, and a name.

```
urlpatterns = [
```

```

url(r'^$', views.index, name='index'),
url(r'^books/$', views.BookListView.as_view(), name='books'),
url(r'^book/(?P<pk>\d+)$', views.BookDetailView.as_view(), name='book-
detail'),
]

```

Unlike our previous mappers, in this case we are using our regular expression (RE) to match against a real "pattern" rather than just a string. What this particular RE does is match against any URL that starts with `book/`, followed by one or more *digits* (numbers) before the end of line marker. While performing the matching, it "captures" the digits, and passes them to the view function as a parameter named `pk`.

Note: As discussed previously, our matched URL is actually `catalog/book/<digits>` (because we are in the **catalog** application, `/catalog/` is assumed).

Important: The generic class-based detail view *expects* to be passed a parameter named `pk`. If you're writing our own function view you can use whatever parameter name you like, or indeed pass the information in an unnamed argument.

A lightning regular expression primer

[Regular expressions](#) are an incredibly powerful pattern mapping tool. We've avoided saying much about them before now because we were matching hard-coded strings in our URLs (rather than patterns), and because they are, frankly, quite unintuitive and scary for beginners.

Note: Don't panic! The sorts of patterns we'll be matching against are quite simple, and in many cases are well documented!

The first thing to know is that regular expressions should usually be declared using the raw string literal syntax (i.e. they are enclosed as shown: `r'<your regular expression text goes here>'`).

The main parts of the syntax you will need to know for declaring the pattern matches are:

Symbol	Meaning
<code>^</code>	Match the beginning of the text
<code>\$</code>	Match the end of the text
<code>\d</code>	Match a digit (0, 1, 2, ... 9)
<code>\w</code>	Match a word character, e.g. any upper- or lower-case character in the alphabet, digit or the underscore character (<code>_</code>)
<code>+</code>	Match one or more of the preceding character. For example, to match one or more digits you would use <code>\d+</code> . To match one or more "a" characters, you could use <code>a+</code>
<code>*</code>	Match zero or more of the preceding character. For example, to match nothing or a word you could use <code>\w*</code>

Symbol	Meaning
()	Capture the part of the pattern inside the brackets. Any captured values will be passed to the view as unnamed parameters (if multiple patterns are captured, the associated parameters will be supplied in the order that the captures were declared).
(?P<name>...)	Capture the pattern (indicated by ...) as a named variable (in this case "name"). The captured values are passed to the view with the name specified. Your view must therefore declare an argument with the same name!
[]	Match against one character in the set. For example, [abc] will match on 'a' or 'b' or 'c'. [-\w] will match on the '-' character or any word character.

Most other characters can be taken literally!

Lets consider a few real examples of patterns:

Pattern	Description
	This is the RE used in our url mapper. It matches a string that has <code>book/</code> at the start of the line (<code>^book/</code>), then has one or more digits (<code>\d+</code>), and then ends (with no non-digit characters before the end of line marker).
<code>r'^book/(?P<pk>\d+)\$'</code>	It also captures all the digits (<code>?P<pk>\d+</code>) and passes them to the view in a parameter named 'pk'. The captured values are always passed as a string!
	For example, this would match <code>book/1234</code> , and send a variable <code>pk='1234'</code> to the view.
<code>r'^book/(\d+)\$'</code>	This matches the same URLs as the preceding case. The captured information would be sent as an unnamed argument to the view.
	This matches a string that has <code>book/</code> at the start of the line (<code>^book/</code>), then has one or more characters that are <i>either</i> a '-' or a word character (<code>[-\w]+</code>), and then ends. It also captures this set of characters and passes them to the view in a parameter named 'stub'.
<code>r'^book/(?P<stub>[-\w]+)\$'</code>	This is a fairly typical pattern for a "stub". Stubs are URL-friendly word-based primary keys for data. You might use a stub if you wanted your book URL to be more informative. For example <code>/catalog/book/the-secret-garden</code> rather than <code>/catalog/book/33</code> .

You can capture multiple patterns in the one match, and hence encode lots of different information in a URL.

Note: As a challenge, consider how you might encode an url to list all books released in a particular year, month, day, and the RE that could be used to match it.

One feature that we haven't used here, but which you may find valuable, is that you can declare and pass [additional options](#) to the view. The options are declared as a dictionary that you pass as the third un-named argument to the `url()` function. This approach can be useful if you want to use the same view for multiple resources, and pass data to configure its behaviour in each case (below we supply a different template in each case).

```
url(r'^/url/$', views.my_reused_view, {'my_template_name': 'some_path'},
name='aurl'),
url(r'^/anotherurl/$', views.my_reused_view, {'my_template_name':
'another_path'}, name='anotherurl'),
```

Note: Both extra options and named captured patterns are passed to the view as *named* arguments. If you use the **same name** for both a captured pattern and an extra option then only the captured pattern value will be sent to the view (the value specified in the additional option will be dropped).

[View \(class-based\)](#)

Open **catalog/views.py**, and copy the following code into the bottom of the file:

```
class BookDetailView(generic.DetailView):
    model = Book
```

That's it! All you need to do now is create a template called **/locallibrary/catalog/templates/catalog/book_detail.html**, and the view will pass it the database information for the specific `Book` record extracted by the URL mapper. Within the template you can access the list of books with the template variable named `object` OR `book` (i.e. generically "*the_model_name*").

If you need to, you can change the template used and the name of the context object used to reference the book in the template. You can also override methods to, for example, add additional information to the context.

[What happens if the record doesn't exist?](#)

If a requested record does not exist then the generic class-based detail view will raise an `Http404` exception for you automatically — in production this will automatically display an appropriate "resource not found" page, which you can customise if desired.

Just to give you some idea of how this works, the code fragment below demonstrates how you would implement the class-based view as a function, if you were **not** using the generic class-based detail view.

```
def book_detail_view(request, pk):
    try:
        book_id=Book.objects.get(pk=pk)
    except Book.DoesNotExist:
        raise Http404("Book does not exist")

    #book_id=get_object_or_404(Book, pk=pk)
```

```

return render(
    request,
    'catalog/book_detail.html',
    context={'book':book_id,}
)

```

The view first tries to get the specific book record from the model. If this fails the view should raise an `Http404` exception to indicate that the book is "not found". The final step is then, as usual, to call `render()` with the template name and the book data in the `context` parameter (as a dictionary).

Note: The `get_object_or_404()` (shown commented out above) is a convenient shortcut to raise an `Http404` exception if the record is not found.

Creating the Detail View template

Create the HTML file `/locallibrary/catalog/templates/catalog/book_detail.html` and give it the below content. As discussed above, this is the default template file name expected by the generic class-based *detail* view (for a model named `Book` in an application named `catalog`).

```

{% extends "base_generic.html" %}

{% block content %}
    <h1>Title: {{ book.title }}</h1>

    <p><strong>Author:</strong> <a href="">{{ book.author }}</a></p> <!--
author detail link not yet defined -->
    <p><strong>Summary:</strong> {{ book.summary }}</p>
    <p><strong>ISBN:</strong> {{ book.isbn }}</p>
    <p><strong>Language:</strong> {{ book.language }}</p>
    <p><strong>Genre:</strong> {% for genre in book.genre.all %} {{ genre }}{%
if not forloop.last %}, {% endif %}{% endfor %}</p>

    <div style="margin-left:20px;margin-top:20px">
        <h4>Copies</h4>

        {% for copy in book.bookinstance_set.all %}
        <hr>
        <p class="{% if copy.status == 'a' %}text-success{% elif copy.status ==
'd' %}text-danger{% else %}text-warning{% endif %}">{{
copy.get_status_display }}</p>
        {% if copy.status != 'a' %}<p><strong>Due to be returned:</strong>
{{copy.due_back}}</p>{% endif %}
        <p><strong>Imprint:</strong> {{copy.imprint}}</p>
        <p class="text-muted"><strong>Id:</strong> {{copy.id}}</p>
        {% endfor %}
    </div>
{% endblock %}

```

The author link in the template above has an empty URL because we've not yet created an author detail page. Once that exists, you should update the URL like this:

```

<a href="{% url 'author-detail' book.author.pk %}">{{ book.author }}</a>

```

Though a little larger, almost everything in this template has been described previously:

- We extend our base template and override the "content" block.
- We use conditional processing to determine whether or not to display specific content.
- We use `for` loops to loop through lists of objects.
- We access the context fields using the dot notation (because we've used the detail generic view, the context is named `book`; we could also use "object")

The one interesting thing we haven't seen before is the function

`book.bookinstance_set.all()`. This method is "automagically" constructed by Django in order to return the set of `BookInstance` records associated with a particular `Book`.

```
{% for copy in book.bookinstance_set.all %}
<!-- code to iterate across each copy/instance of a book -->
{% endfor %}
```

This method is needed because you declare a `ForeignKey` (one-to many) field in only the "one" side of the relationship. Since you don't do anything to declare the relationship in the other ("many") model, it doesn't have any field to get the set of associated records. To overcome this problem, Django constructs an appropriately named "reverse lookup" function that you can use. The name of the function is constructed by lower-casing the model name where the `ForeignKey` was declared, followed by `_set` (i.e. so the function created in `Book` is `bookinstance_set()`).

Note: Here we use `all()` to get all records (the default). While you can use the `filter()` method to get a subset of records in code, you can't do this directly in templates because you can't specify arguments to functions.

What does it look like?[Edit](#)

At this point we should have created everything needed to display both the book list and book detail pages. Run the server (`python3 manage.py runserver`) and open your browser to <http://127.0.0.1:8000/>.

Warning: Don't click any author or author detail links yet — you'll create those in the challenge!

Click the **All books** link to display the list of books.



Then click a link to one of your books. If everything is set up correctly, you should see something like the following screenshot.

[Home](#)
[All books](#)
[All authors](#)

Title: The Name of the Wind (The Kingkiller Chronicle, #1)

Author: [Rothfuss, Patrick](#)

Summary: Told in Kvothe's own voice, this is the tale of the magically gifted young man who grows to be the most notorious wizard his world has ever seen.

ISBN: 9780756404079

Language: English

Genre: Fantasy

Copies

Available

Imprint: Published March 27th 2007 by Penguin Group DAW Hardcover

Id: 344edc33-37a0-497f-b574-637b6ddcf285

Maintenance

Due to be returned: Oct. 12, 2016

Imprint: Published March 27th 2007 by Penguin Group DAW Hardcover

Id: 99841f53-c533-494b-b33d-687946f19bed

On loan

Due to be returned: Oct. 12, 2016

Imprint: Published March 27th 2007 by Penguin Group DAW Hardcover

Id: 8edafcc6-9d1e-4056-9f20-ce99d1ef157e

Pagination [Edit](#)

If you've just got a few records, our book list page will look fine. However, as you get into the tens or hundreds of records the page will take progressively longer to load (and have far too much content to browse sensibly). The solution to this problem is to add pagination to your list views, reducing the number of items displayed on each page.

Django has excellent in-built support for pagination. Even better, this is built into the generic class-based list views so you don't have to do very much to enable it!

Views

Open **catalog/views.py**, and add the `paginate_by` line shown in bold below.

```
class BookListView(generic.ListView):  
    model = Book  
    paginate_by = 10
```

With this addition, as soon as you have more than 10 records the view will start paginating the data it sends to the template. The different pages are accessed using GET parameters — to access page 2 you would use the URL: `/catalog/books/?page=2`.

Templates

Now that the data is paginated, we need to add support to the template to scroll through the results set. Because we might want to do this in all list views, we'll do this in a way that can be added to the base template.

Open `/locallibrary/catalog/templates/base_generic.html` and copy in the following pagination block below our content block (highlighted below in bold). The code first checks if pagination is enabled on the current page. If so then it adds next and previous links as appropriate (and the current page number).

```
{% block content %}{% endblock %}

{% block pagination %}
    {% if is_paginated %}
        <div class="pagination">
            <span class="page-links">
                {% if page_obj.has_previous %}
                    <a href="{{ request.path }}?page={{
page_obj.previous_page_number }}">previous</a>
                {% endif %}
                <span class="page-current">
                    Page {{ page_obj.number }} of {{
page_obj.paginator.num_pages }}.
                </span>
                {% if page_obj.has_next %}
                    <a href="{{ request.path }}?page={{
page_obj.next_page_number }}">next</a>
                {% endif %}
            </span>
        </div>
    {% endif %}
{% endblock %}
```

The `page_obj` is a [Paginator](#) object that will exist if pagination is being used on the current page. It allows you to get all the information about the current page, previous pages, how many pages there are, etc.

We use `{{ request.path }}` to get the current page URL for creating the pagination links. This is useful, because it is independent of the object that we're paginating.

Thats it!

What does it look like?

The screenshot below shows what the pagination looks like — if you haven't entered more than 10 titles into your database, then you can test it more easily by lowering the number specified in the `paginate_by` line in your `catalog/views.py` file. To get the below result we changed it to `paginate_by = 2`.

The pagination links are displayed on the bottom, with next/previous links being displayed depending on which page you're on.



Challenge yourself [Edit](#)

The challenge in this article is to create the author detail and list views required to complete the project. These should be made available at the following URLs:

- `catalog/authors/` — The list of all authors.
- `catalog/author/<id>` — The detail view for the specific author with a primary key field named `<id>`

The code required for the URL mappers and the views should be virtually identical to the `Book` list and detail views we created above. The templates will be different, but will share similar behaviour.

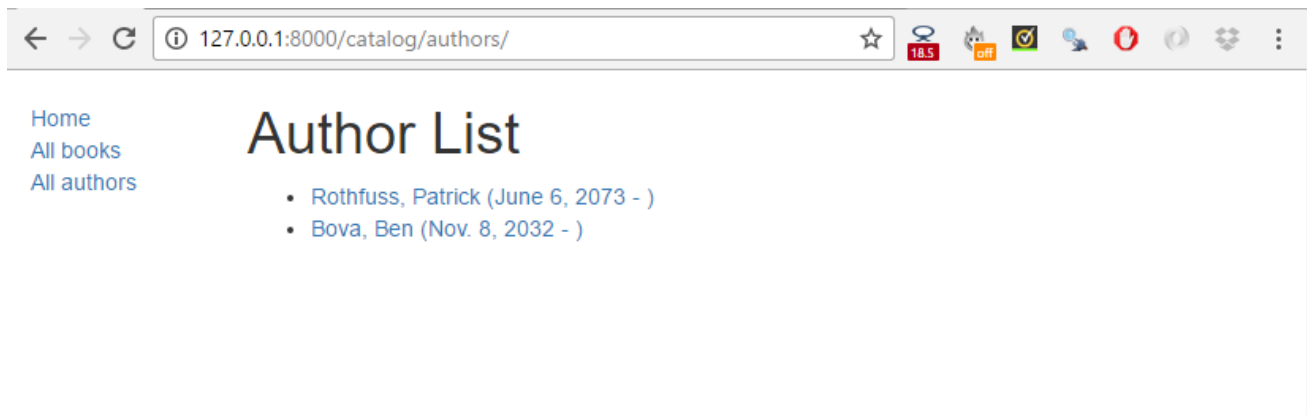
Note:

- Once you've created the URL mapper for the author list page you will also need to update the **All authors** link in the base template. Follow the [same process](#) as we did when we updated the **All books** link.
- Once you've created the URL mapper for the author detail page, you should also update the [book detail view template](#) (`/locallibrary/catalog/templates/catalog/book_detail.html`) so that the author link points to your new author detail page (rather than being an empty URL). The line will change to add the template tag shown in bold below.

```
<p><strong>Author:</strong> <a href="{% url 'author-detail'
book.author.pk %}">{{ book.author }}</a></p>
```

-

When you are finished, your pages should look something like the screenshots below.



[Summary](#)[Edit](#)

Congratulations, our basic library functionality is now complete!

In this article we've learned how to use the generic class-based list and detail views and used them to create pages to view our books and authors. Along the way we've learned about pattern matching with regular expressions, and how you can pass data from URLs to your views. We've also learned a few more tricks for using templates. Last of all we've shown how to paginate list views, so that our lists are manageable even when we have many records.

Django Tutorial Part 7: Sessions framework

This tutorial extends our [LocalLibrary](#) website, adding a session-based visit-counter to the home page. This is a relatively simple example, but it does show how you can use the session framework to provide persistent behaviour for anonymous users in your own sites.

Prerequisites: Complete all previous tutorial topics, including [Django Tutorial Part 6: Generic list and detail views](#)

Objective: To understand how sessions are used.

[Overview](#)[Edit](#)

The [LocalLibrary](#) website we created in the previous tutorials allows users to browse books and authors in the catalog. While the content is dynamically generated from the database, every user will essentially have access to the same pages and types of information when they use the site.

In a "real" library you may wish to provide individual users with a customised experience, based on their previous use of the site, preferences, etc. For example, you could hide warning messages that the user has previously acknowledged next time they visit the site, or store and respect their preferences (e.g. the number of search results they want displayed on each page).

The session framework lets you implement this sort of behaviour, allowing you to store and retrieve arbitrary data on a per-site-visitor basis.

[What are sessions?](#)[Edit](#)

All communication between web browsers and servers is via the HTTP protocol, which is *stateless*. The fact that the protocol is stateless means that messages between the client and server are completely independent of each other—there is no notion of "sequence" or behaviour based on previous messages. As a result, if you want to have a site that keeps track of the ongoing relationships with a client, you need to implement that yourself.

Sessions are the mechanism used by Django (and most of the Internet) for keeping track of the "state" between the site and a particular browser. Sessions allow you to store arbitrary data per browser, and have this data available to the site whenever the browser connects. Individual data items associated with the session are then referenced by a "key", which is used both to store and retrieve the data.

Django uses a cookie containing a special *session id* to identify each browser and its associated session with the site. The actual session *data* is stored in the site database by default (this is more secure than storing the data in a cookie, where they are more vulnerable to malicious users). You can configure Django to store the session data in other places (cache, files, "secure" cookies), but the default location is a good and relatively secure option.

[Enabling sessions](#)[Edit](#)

Sessions were enabled automatically when we [created the skeleton website](#) (in tutorial 2).

The configuration is set up in the `INSTALLED_APPS` and `MIDDLEWARE` sections of the project file (`locallibrary/locallibrary/settings.py`), as shown below:

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',
    ....

MIDDLEWARE = [
    ...
    'django.contrib.sessions.middleware.SessionMiddleware',
    ....
```

[Using sessions](#)[Edit](#)

You can access the `session` attribute in the view from the `request` parameter (an `HttpRequest` passed in as the first argument to the view). This session attribute represents the specific connection to the current user (or to be more precise, the connection to the current *browser*, as identified by the session id in the browser's cookie for this site).

The `session` attribute is a dictionary-like object that you can read and write as many times as you like in your view, modifying it as wished. You can do all the normal dictionary operations, including clearing all data, testing if a key is present, looping through data, etc. Most of the time though, you'll just use the standard "dictionary" API to get and set values.

The code fragments below show how you can get, set, and delete some data with the key `"my_car"`, associated with the current session (browser).

Note: One of the great things about Django is that you don't need to think about the mechanisms that tie the session to your current request in your view. If we were to use the fragments below in our view, we'd know that the information about `my_car` is associated only with the browser that sent the current request.

```
# Get a session value by its key (e.g. 'my_car'), raising a KeyError if the
key is not present
my_car = request.session['my_car']

# Get a session value, setting a default if it is not present ('mini')
my_car = request.session.get('my_car', 'mini')

# Set a session value
request.session['my_car'] = 'mini'

# Delete a session value
del request.session['my_car']
```

The API also offers a number of other methods that are mostly used to manage the associated session cookie. For example, there are methods to test that cookies are supported in the client browser, to set and check cookie expiry dates, and to clear expired sessions from the data store. You can find out about the full API in [How to use sessions](#) (Django docs).

Saving session data [Edit](#)

By default, Django only saves to the session database and sends the session cookie to the client when the session has been *modified* (assigned) or *deleted*. If you're updating some data using its session key as shown in the previous section, then you don't need to worry about this! For example:

```
# This is detected as an update to the session, so session data is saved.
request.session['my_car'] = 'mini'
```

If you're updating some information *within* session data, then Django will not recognise that you've made a change to the session and save the data (for example, if you were to change "wheels" data inside your "my_car" data, as shown below). In this case you will need to explicitly mark the session as having been modified.

```
# Session object not directly modified, only data within the session. Session
changes not saved!
request.session['my_car']['wheels'] = 'alloy'

# Set session as modified to force data updates/cookie to be saved.
request.session.modified = True
```

Note: You can change the behavior so the site will update the database/send cookie on every request by adding `SESSION_SAVE_EVERY_REQUEST = True` into your project settings (`locallibrary/locallibrary/settings.py`).

Simple example — getting visit counts [Edit](#)

As a simple real-world example we'll update our library to tell the current user how many times they have visited the *LocalLibrary* home page.

Open `/locallibrary/catalog/views.py`, and make the changes shown in bold below.

```
def index(request):
    ...

    num_authors=Author.objects.count() # The 'all()' is implied by default.

    # Number of visits to this view, as counted in the session variable.
    num_visits=request.session.get('num_visits', 0)
    request.session['num_visits'] = num_visits+1

    # Render the HTML template index.html with the data in the context
    variable.
    return render(
        request,
        'index.html',

        context={'num_books':num_books,'num_instances':num_instances,'num_instances_a
        vailable':num_instances_available,'num_authors':num_authors,
        'num_visits':num_visits}, # num_visits appended
    )
```

Here we first get the value of the 'num_visits' session key, setting the value to 0 if it has not previously been set. Each time a request is received, we then increment the value and store it back in the session (for the next time the user visits the page). The num_visits variable is then passed to the template in our context variable.

Note: We might also test whether cookies are even supported in the browser here (see [How to use sessions](#) for examples) or design our UI so that it doesn't matter whether or not cookies are supported.

Add the line seen at the bottom of the following block to your main HTML template (**/locallibrary/catalog/templates/index.html**) at the bottom of the "Dynamic content" section to display the context variable:

```
<h2>Dynamic content</h2>

<p>The library has the following record counts:</p>
<ul>
<li><strong>Books:</strong> {{ num_books }}</li>
<li><strong>Copies:</strong> {{ num_instances }}</li>
<li><strong>Copies available:</strong> {{ num_instances_available }}</li>
<li><strong>Authors:</strong> {{ num_authors }}</li>
</ul>

<p>You have visited this page {{ num_visits }}{% if num_visits == 1 %} time{%
else %} times{% endif %}.</p>
```

Save your changes and restart the test server. Every time you refresh the page, the number should update.

[Summary](#)[Edit](#)

You now know how easy it is to use sessions to improve your interaction with *anonymous* users.

In our next articles we'll explain the authentication and authorisation (permission) framework, and show you how to support user accounts.

Django Tutorial Part 8: User authentication and permissions

In this tutorial we'll show you how to allow users to login to your site with their own accounts, and how to control what they can do and see based on whether or not they are logged in and their *permissions*. As part of this demonstration we'll extend the [LocalLibrary](#) website, adding login and logout pages, and user- and staff-specific pages for viewing books that have been borrowed.

Prerequisites: Complete all previous tutorial topics, up to and including [Django Tutorial Part 7: Sessions framework](#).

Objective: To understand how to set up and use user authentication and permissions.

[Overview](#)[Edit](#)

Django provides an authentication and authorisation ("permission") system, built on top of the session framework discussed in the [previous tutorial](#), that allows you to verify user credentials and define what actions each user is allowed to perform. The framework includes built-in models for `Users` and `Groups` (a generic way of applying permissions to more than one user at a time), permissions/flags that designate whether a user may perform a task, forms and views for logging in users, and view tools for restricting content.

Note: According to Django the authentication system aims to be very generic, and so does not provide some features provided in other web authentication systems. Solutions for some common problems are available as third party packages. For example, throttling of login attempts and authentication against third parties (e.g. OAuth).

In this tutorial we'll show you how to enable user authentication in the [LocalLibrary](#) website, create your own login and logout pages, add permissions to your models, and control access to pages. We'll use the authentication/permissions to display lists of books that have been borrowed for both users and librarians.

The authentication system is very flexible, and you can build up your URLs, forms, views, and templates from scratch if you like, just calling the provided API to login the user. However, in this article we're going to use Django's "stock" authentication views and forms for our login and logout pages. We'll still need to create some templates, but that's pretty easy.

We'll also show you how to create permissions, and check on login status and permissions in both views and templates.

[Enabling authentication](#)[Edit](#)

The authentication was enabled automatically when we [created the skeleton website](#) (in tutorial 2) so you don't need to do anything more at this point.

Note: The necessary configuration was all done for us when we created the app using the `django-admin startproject` command. The database tables for users and model permissions were created when we first called `python manage.py migrate`.

The configuration is set up in the `INSTALLED_APPS` and `MIDDLEWARE` sections of the project file (`locallibrary/locallibrary/settings.py`), as shown below:

```
INSTALLED_APPS = [
    ...
    'django.contrib.auth', #Core authentication framework and its default
models.
    'django.contrib.contenttypes', #Django content type system (allows
permissions to be associated with models).
    ....

MIDDLEWARE = [
    ...
    'django.contrib.sessions.middleware.SessionMiddleware', #Manages
sessions across requests
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware', #Associates
users with requests using sessions.
    ....
```

Creating users and groups[Edit](#)

You already created your first user when we looked at the [Django admin site](#) in tutorial 4 (this was a superuser, created with the command `python manage.py createsuperuser`). Our superuser is already authenticated and has all permissions, so we'll need to create a test user to represent a normal site user. We'll be using the admin site to create our *locallibrary* groups and website logins, as it is one of the quickest ways to do so.

Note: You can also create users programmatically, as shown below. You would have to do this, for example, if developing an interface to allow users to create their own logins (you shouldn't give users access to the admin site).

```
from django.contrib.auth.models import User

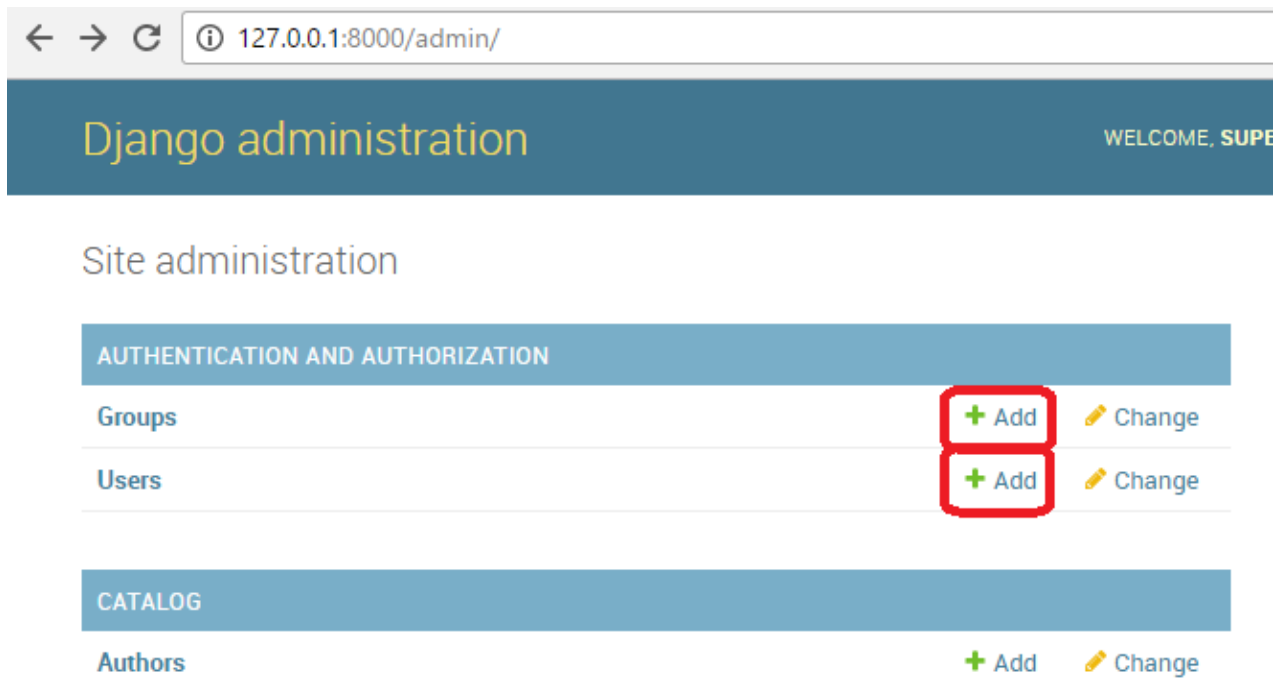
# Create user and save to the database
user = User.objects.create_user('myusername', 'myemail@crazymail.com',
'mypassword')

# Update fields and then save again
user.first_name = 'John'
user.last_name = 'Citizen'
user.save()
```

Below we'll first create a group and then a user. Even though we don't have any permissions to add for our library members yet, if we need to later, it will be much easier to add them once to the group than individually to each member.

Start the development server and navigate to the admin site in your local web browser (<http://127.0.0.1:8000/admin/>). Login to the site using the credentials for your superuser account. The top level of the Admin site displays all of your models, sorted by "django application". From

the **Authentication and Authorisation** section you can click the **Users** or **Groups** links to see their existing records.



First lets create a new group for our library members.

1. Click the **Add** button (next to Group) to create a new *Group*; enter the **Name** "Library Members" for the group.

The screenshot shows the 'Add group' form. The breadcrumb trail at the top reads 'Home > Authentication and Authorization > Groups > Add group'. The form title is 'Add group'. The 'Name' field is a text input containing 'Library Members' (highlighted with a red box). Below the name field, the 'Permissions' section is divided into two panes: 'Available permissions' and 'Chosen permissions'. The 'Available permissions' pane has a search filter and a list of permissions, including 'admin | log entry | Can add log entry', 'admin | log entry | Can change log entry', 'admin | log entry | Can delete log entry', 'auth | group | Can add group', 'auth | group | Can change group', 'auth | group | Can delete group', 'auth | permission | Can add permission', 'auth | permission | Can change permission', 'auth | permission | Can delete permission', 'auth | user | Can add user', 'auth | user | Can change user', and 'auth | user | Can delete user'. The 'Chosen permissions' pane is currently empty. At the bottom of the form, there are three buttons: 'Save and add another', 'Save and continue editing', and 'SAVE' (highlighted with a red box). A note at the bottom states: 'Hold down "Control", or "Command" on a Mac, to select more than one.'

2. We don't need any permissions for the group, so just press **SAVE** (you will be taken to a list of groups).

Now lets create a user:

1. Navigate back to the home page of the admin site
2. Click the **Add** button next to *Users* to open the *Add user* dialog.

The screenshot shows the Django administration interface for adding a new user. At the top, the header bar displays 'Django administration' on the left and 'WELCOME, SUPERMAN. VIEW SITE / CHANGE PASSWORD / LOG OUT' on the right. Below the header, a breadcrumb trail reads 'Home > Authentication and Authorization > Users > Add user'. The main heading is 'Add user'. A subtext instruction says: 'First, enter a username and password. Then, you'll be able to edit more user options.' The form contains three input fields: 'Username:' with the value 'albertu' and a note 'Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.'; 'Password:' with masked characters; and 'Password confirmation:' with masked characters and a note 'Enter the same password as before, for verification.' At the bottom right, there are three buttons: 'Save and add another', 'Save and continue editing', and 'SAVE'.

3. Enter an appropriate **Username** and **Password/Password confirmation** for your test user
4. Press **SAVE** to create the user.

The admin site will create the new user and immediately take you to a *Change user* screen where you can change your **username** and add information for the User model's optional fields. These fields include the first name, last name, email address, the users status and permissions (only the **Active** flag should be set). Further down you can specify the user's groups and permissions, and see important dates related to the user (e.g. their join date and last login

Home » Authentication and Authorization » Users » albertu

✔ The user "albertu" was added successfully. You may edit it again below.

HISTORY

albertu

```
algorithm: pbkdf2_sha256 iterations: 30000 salt: 7Rpnok***** hash: rXmew+*****
```

Designates that this user has all permissions without explicitly assigning them.

The screenshot shows two side-by-side panels. The left panel, titled 'Available groups', contains a search bar with the placeholder text 'Filter' and a 'Choose all' button at the bottom. The right panel, titled 'Chosen groups', contains a 'Library Members' section and a 'Remove all' button at the bottom. A red box highlights the arrow buttons on the right side of the 'Available groups' panel.

That's it!. You now you have a "normal library member" account that you will be able to use for testing (once we've implemented the pages to enable them to login).

Note: You should try creating another library member user. Also, create a group for Librarians, and add a user to that too!

Setting up your authentication views [Edit](#)

Django's provides almost everything you need to create authentication pages to handle login, logout, and password management "out of the box". This includes an url mapper, views and forms, but it does not include the templates — we have to create our own!

In this section we show how to integrate the default system into the *LocalLibrary* website and create the templates. We'll put them in the main project URLs.

Note: You don't have to use any of this code, but it is likely that you'll want to because it makes things a lot easier. You'll almost certainly need to change the form handling code if you change your user model (an advanced topic!) but even so, you would still be able to use the stock view functions.

Note: In this case we could reasonably put the authentication pages, including the URLs and templates, inside our catalog application. However if we had multiple applications it would be better to separate out this shared login behaviour and have it available across the whole site, so that is what we've shown here!

Project URLs

Add the following to the bottom of the project urls.py file (**locallibrary/locallibrary/urls.py**) file:

```
#Add Django site authentication urls (for login, logout, password management)
urlpatterns += [
    url(r'^accounts/', include('django.contrib.auth.urls')),
]
```

Navigate to the <http://127.0.0.1:8000/accounts/> URL (note the trailing forward slash!) and Django will show an error that it could not find this URL, and listing all the URLs it tried. From this you can see the URLs that will work, for example:

```
^accounts/ ^login/$ [name='login']
^accounts/ ^logout/$ [name='logout']
^accounts/ ^password_change/$ [name='password_change']
^accounts/ ^password_change/done/$ [name='password_change_done']
^accounts/ ^password_reset/$ [name='password_reset']
^accounts/ ^password_reset/done/$ [name='password_reset_done']
^accounts/ ^reset/(?P<uidb64>[0-9A-Za-z_-]+)/ (?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]{1,20})/$ [name='password_reset_confirm']
^accounts/ ^reset/done/$ [name='password_reset_complete']
```

Now try to navigate to the login URL (<http://127.0.0.1:8000/accounts/login/>). This will fail again, but with an error that tells you that we're missing the required template (**registration/login.html**) on the template search path. You'll see the following lines listed in the yellow section up the top:

```
Exception Type:      TemplateDoesNotExist
```

Exception Value: **registration/login.html**

The next step is to create a registration directory on the search path and then add the **login.html** file.

Template directory

The urls (and implicitly views) that we just added expect to find their associated templates in a directory **/registration/** somewhere in the templates search path.

For this site we'll put our HTML pages in the **templates/registration/** directory. This directory should be in your project root directory, i.e the same directory as as the **catalog** and **locallibrary** folders). Please create these folders now.

Note: Your folder structure should now look like the below:

/locallibrary (django project folder)

 /catalog

 /locallibrary

 /templates (new)

 /registration

To make these directories visible to the template loader (i.e. to put this directory in the template search path) open the project settings (**/locallibrary/locallibrary/settings.py**), and update the **TEMPLATES** section's **'DIRS'** line as shown.

```
TEMPLATES = [  
    {  
        ...  
        'DIRS': ['./templates'],  
        'APP_DIRS': True,  
        ...  
    },  
]
```

Login template

Important: The authentication templates provided in this article are very basic/slightly modified version of the Django demonstration login templates. You may need to customise them for your own use!

Create a new HTML file called **/locallibrary/templates/registration/login.html**. give it the following contents:

```
{% extends "base_generic.html" %}  
  
{% block content %}  
  
{% if form.errors %}  
<p>Your username and password didn't match. Please try again.</p>  
{% endif %}  
  
{% if next %}  
  {% if user.is_authenticated %}  
  <p>Your account doesn't have access to this page. To proceed,  
  please login with an account that has access.</p>  
  {% else %}
```

```

        <p>Please login to see this page.</p>
        {% endif %}
    {% endif %}

<form method="post" action="{% url 'login' %}">
    {% csrf_token %}

    <div>
        <td>{{ form.username.label_tag }}</td>
        <td>{{ form.username }}</td>
    </div>
    <div>
        <td>{{ form.password.label_tag }}</td>
        <td>{{ form.password }}</td>
    </div>

    <div>
        <input type="submit" value="login" />
        <input type="hidden" name="next" value="{{ next }}" />
    </div>
</form>

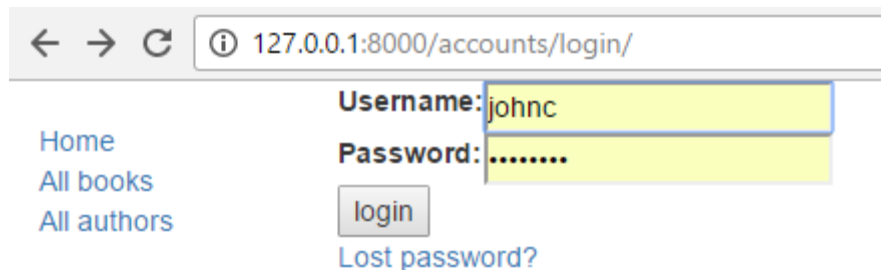
{# Assumes you setup the password_reset view in your URLconf #}
<p><a href="{% url 'password_reset' %}">Lost password?</a></p>

{% endblock %}

```

This template shares some similarities with the ones we've seen before — it extends our base template and overrides the `content` block. The rest of the code is fairly standard form handling code, which we will discuss in a later tutorial. All you need to know for now is that this will display a form in which you can enter your username and password, and that if you enter invalid values you will be prompted to enter correct values when the page refreshes.

Navigate back to the login page (<http://127.0.0.1:8000/accounts/login/>) once you've saved your template, and you should see something like this:



If you try to login that will succeed and you'll be redirected to another page (by default this will be <http://127.0.0.1:8000/accounts/profile/>). The problem here is that by default Django expects that after login you will want to be taken to a profile page, which may or may not be the case. As you haven't defined this page yet, you'll get another error!

Open the project settings (`/locallibrary/locallibrary/settings.py`) and add the text below to the bottom. Now when you login you should be redirected to the site home page by default.

```
# Redirect to home URL after login (Default redirects to /accounts/profile/)
LOGIN_REDIRECT_URL = '/'
```

Logout template

If you navigate to the logout url (<http://127.0.0.1:8000/accounts/logout/>) then you'll see some odd behaviour — your user will be logged out sure enough, but you'll be taken to the **Admin** logout page. That's not what you want, if only because the login link on that page takes you to the Admin login screen (and that is only available to users who have the `is_staff` permission).

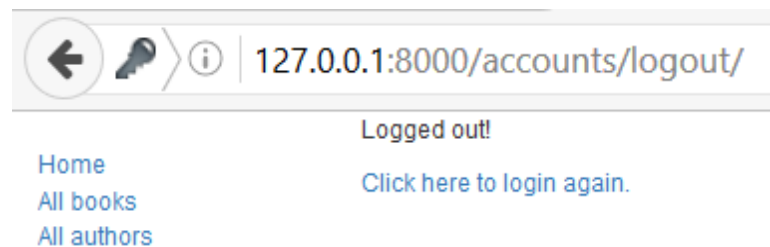
Create and open `/locallibrary/templates/registration/logged_out.html`. Copy in the text below:

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

{% block content %}
<p>Logged out!</p>

<a href="{% url 'login'%}">Click here to login again.</a>
{% endblock %}
```

This template is very simple. It just displays a message informing you that you have been logged out, and provides a link that you can press to go back to the login screen. If you go to the logout URL again you should see this page:



Password reset templates

The default password reset system uses email to send the user a reset link. You need to create forms to get the user's email address, send the email, allow them to enter a new password, and to note when the whole process is complete.

The following templates can be used as a starting point.

Password reset form

This is the form used to get the user's email address (for sending the password reset email). Create `/locallibrary/templates/registration/password_reset_form.html`, and give it the following contents:

```
{% extends "base_generic.html" %}
{% block content %}

<form action="" method="post">{% csrf_token %}
```

```

        {% if form.email.errors %} {{ form.email.errors }} {% endif %}
        <p>{{ form.email }}</p>
        <input type="submit" class="btn btn-default btn-lg" value="Reset
password" />
</form>

{% endblock %}
Password reset done

```

This form is displayed after your email address has been collected. Create **/locallibrary/templates/registration/password_reset_done.html**, and give it the following contents:

```

{% extends "base_generic.html" %}
{% block content %}
<p>We've emailed you instructions for setting your password. If they haven't
arrived in a few minutes, check your spam folder.</p>
{% endblock %}
Password reset email

```

This template provides the text of the HTML email we will sent to users, which contains the reset link. Create **/locallibrary/templates/registration/password_reset_email.html**, and give it the following contents:

```

Someone asked for password reset for email {{ email }}. Follow the link
below:
{{ protocol }}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid
token=token %}
Password reset confirm

```

This page is where you enter your new password after clicking the link in the password-reset email. Create **/locallibrary/templates/registration/password_reset_confirm.html**, and give it the following contents:

```

{% extends "base_generic.html" %}

{% block content %}

    {% if validlink %}
        <p>Please enter (and confirm) your new password.</p>
        <form action="" method="post">
            <div style="display:none">
                <input type="hidden" value="{{ csrf_token }}"
name="csrfmiddlewaretoken">
            </div>
            <table>
                <tr>
                    <td>{{ form.new_password1.errors }}
                        <label for="id_new_password1">New
password:</label></td>
                    <td>{{ form.new_password1 }}</td>
                </tr>
                <tr>
                    <td>{{ form.new_password2.errors }}
                        <label for="id_new_password2">Confirm
password:</label></td>

```

```

        <td>{{ form.new_password2 }}</td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit" value="Change my password"
/></td>
    </tr>
</table>
</form>
{% else %}
    <h1>Password reset failed</h1>
    <p>The password reset link was invalid, possibly because it has
already been used. Please request a new password reset.</p>
{% endif %}

{% endblock %}

```

Password reset complete

This is the last password-reset template, which is displayed to notify you when the password reset has succeeded. Create **/locallibrary/templates/registration/password_reset_complete.html**, and give it the following contents:

```

{% extends "base_generic.html" %}
{% block content %}

<h1>The password has been changed!</h1>
<p><a href="{% url 'login' %}">log in again?</a></p>

{% endblock %}

```

Testing the new authentication pages

Now that you've added the URL configuration and created all these templates, the authentication pages should now just work!

You can test the new authentication pages by attempting to login and then logout your superuser account using these URLs:

- <http://127.0.0.1:8000/accounts/login/>
- <http://127.0.0.1:8000/accounts/logout/>

You'll be able to test the password reset functionality from the link in the login page. **Be aware that Django will only send reset emails to addresses (users) that are already stored in its database!**

Note: The password reset system requires that your website supports email, which is beyond the scope of this article, so this part **won't work yet**. To allow testing, put the following line at the end of your settings.py file. This logs any emails sent to the console (so you can copy the password reset link from the console).

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

For more information, see [Sending email](#) (Django docs).

Testing against authenticated users [Edit](#)

This section looks at what we can do to selectively control content the user sees based on whether they are logged in or not.

Testing in templates

You can get information about the currently logged in user in templates with the `{{ user }}` template variable (this is added to the template context by default when you set up the project as we did in our skeleton).

Typically you will first test against the `{{ user.is_authenticated }}` template variable to determine whether the user is eligible to see specific content. To demonstrate this, next we'll update our sidebar to display a "Login" link if the user is logged out, and a "Logout" link if they are logged in.

Open the base template (`/locallibrary/catalog/templates/base_generic.html`) and copy the following text into the `sidebar` block, immediately before the `endblock` template tag.

```
<ul class="sidebar-nav">

    ...

    {% if user.is_authenticated %}
        <li>User: {{ user.get_username }}</li>
        <li><a href="{% url 'logout' %}?next={{ request.path }}">Logout</a></li>
    {% else %}
        <li><a href="{% url 'login' %}?next={{ request.path }}">Login</a></li>
    {% endif %}
</ul>
```

As you can see, we use `if-else-endif` template tags to conditionally display text based on whether `{{ user.is_authenticated }}` is true. If the user is authenticated then we know that we have a valid user, so we call `{{ user.get_username }}` to display their name.

We create the login and logout link URLs using the `url` template tag and the names of the respective URL configurations. Note also how we have appended `?next={{ request.path }}` to the end of the URLs. What this does is add a URL parameter `next` containing the address (URL) of the *current* page, to the end of the linked URL. After the user has successfully logged in/out, the views will use this "next" value to redirect the user back to the page where they first clicked the login/logout link.

Note: Try it out! If you're on the home page and you click Login/Logout in the sidebar, then after the operation completes you should end up back on the same page.

Testing in views

If you're using function-based views, the easiest way to restrict access to your functions is to apply the `login_required` decorator to your view function, as shown below. If the user is logged in then your view code will execute as normal. If the user is not logged in, this will redirect to the login URL defined in the project settings (`settings.LOGIN_URL`), passing the

current absolute path as the `next` URL parameter. If the user succeeds in logging in then they will be returned back to this page, but this time authenticated.

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

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

Note: You can do the same sort of thing manually by testing `on request.user.is_authenticated`, but the decorator is much more convenient!

Similarly, the easiest way to restrict access to logged-in users in your class-based views is to derive from `LoginRequiredMixin`. You need to declare this mixin first in the super class list, before the main view class.

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

class MyView(LoginRequiredMixin, View):
    ...
```

This has exactly the same redirect behaviour as the `login_required` decorator. You can also specify an alternative location to redirect the user to if they are not authenticated (`login_url`), and a URL parameter name instead of "next" to insert the current absolute path (`redirect_field_name`).

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

For additional detail, check out the [Django docs here](#).

[Example — listing the current user's books](#)[Edit](#)

Now that we know how to restrict a page to a particular user, lets create a view of the books that the current user has borrowed.

Unfortunately we don't yet have any way for users to borrow books! So before we can create the book list we'll first extend the `BookInstance` model to support the concept of borrowing and use the Django Admin application to loan a number of books to our test user.

Models

First we're going to have to make it possible for users to have a `BookInstance` on loan (we already have a `status` and a `due_back` date, but we don't yet have any association between this model and a `User`. We'll create one using a `ForeignKey` (one-to-many) field. We also need an easy mechanism to test whether a loaned book is overdue.

Open `catalog/models.py`, and import the `User` model from `django.contrib.auth.models` (add this just below the previous import line at the top of the file, so `User` is available to subsequent code that makes use of it):

```
from django.contrib.auth.models import User
```

Next add the `borrower` field to the `BookInstance` model:

```
borrower = models.ForeignKey(User, on_delete=models.SET_NULL, null=True,
blank=True)
```

While we're here, let's add a property that we can call from our templates to tell if a particular book instance is overdue. While we could calculate this in the template itself, using a property as shown below will be much more efficient.

```
from datetime import date

@property
def is_overdue(self):
    if date.today() > self.due_back:
        return True
    return False
```

Now that we've updated our models, we'll need to make fresh migrations on the project and then apply those migrations:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

[Admin](#)

Now open **catalog/admin.py**, and add the `borrower` field to the `BookInstanceAdmin` class in both the `list_display` and the `fieldsets` as shown below. This will make the field visible in the Admin section, so that we can assign a `User` to a `BookInstance` when needed.

```
@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    list_display = ('book', 'status', 'borrower', 'due_back', 'id')
    list_filter = ('status', 'due_back')

    fieldsets = (
        (None, {
            'fields': ('book', 'imprint', 'id')
        }),
        ('Availability', {
            'fields': ('status', 'due_back', 'borrower',)
        }),
    )
```

[Loan a few books](#)

Now that it's possible to loan books to a specific user, go and loan out a number of `BookInstance` records (set their `borrowed` field to your test user, make the `status` "On loan" and set due dates both in the future and the past.

Note: We won't spell the process out, as you already know how to use the Admin site!

On loan view

Now we'll add a view for getting the list of all books that have been loaned to the current user. We'll use the same generic class-based list view we're familiar with, but this time we'll also import and derive from `LoginRequiredMixin`, so that only a logged in user can call this view. We will also choose to declare a `template_name`, rather than using the default, because we may end up having a few different lists of `BookInstance` records, with different views and templates.

Add the following to `catalog/views.py`:

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

class LoanedBooksByUserListView(LoginRequiredMixin, generic.ListView):
    """
    Generic class-based view listing books on loan to current user.
    """
    model = BookInstance
    template_name = 'catalog/bookinstance_list_borrowed_user.html'
    paginate_by = 10

    def get_queryset(self):
        return
        BookInstance.objects.filter(borrower=self.request.user).filter(status__exact=
        'o').order_by('due_back')
```

In order to restrict our query to just the `BookInstance` objects for the current user, we re-implement `get_queryset()` as shown above. Note that "o" is the stored code for "on loan" and we order by the `due_back` date so that the oldest items are displayed first.

URL conf for on loan books

Now open `/catalog/urls.py` and add an `url()` pointing to the above view (you can just copy the text below to the end of the file).

```
urlpatterns += [
    url(r'^mybooks/$', views.LoanedBooksByUserListView.as_view(), name='my-
    borrowed'),
]
```

Template for on loan books

Now all we need to do for this page is add a template. First, create the template file `/catalog/templates/catalog/bookinstance_list_borrowed_user.html` and give it the following contents:

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

{% block content %}
    <h1>Borrowed books</h1>

    {% if bookinstance_list %}
    <ul>

        {% for bookinst in bookinstance_list %}
        <li class="{% if bookinst.is_overdue %}text-danger{% endif %}">
```

```

        <a href="{% url 'book-detail' bookinst.book.pk
%}">{{bookinst.book.title}}</a> ({{ bookinst.due_back }})
    </li>
    {% endfor %}
</ul>

{% else %}
    <p>There are no books borrowed.</p>
{% endif %}
{% endblock %}

```

This template is very similar to those we've created previously for the `Book` and `Author` objects. The only thing "new" here is that we check the method we added in the model (`bookinst.is_overdue`) and use it to change the colour of overdue items.

When the development server is running, you should now be able to view the list for a logged in user in your browser at <http://127.0.0.1:8000/catalog/mybooks/>. Try this out with your user logged in and logged out (in the second case, you should be redirected to the login page).

Add the list to the sidebar

The very last step is to add a link for this new page into the sidebar. We'll put this in the same section where we display other information for the logged in user.

Open the base template (`/locallibrary/catalog/templates/base_generic.html`) and add the line in bold to the sidebar as shown.

```

<ul class="sidebar-nav">
    {% if user.is_authenticated %}
    <li>User: {{ user.get_username }}</li>
    <li><a href="{% url 'my-borrowed' %}">My Borrowed</a></li>
    <li><a href="{% url 'logout' %}?next={{ request.path }}">Logout</a></li>
    {% else %}
    <li><a href="{% url 'login' %}?next={{ request.path }}">Login</a></li>
    {% endif %}
</ul>

```

What does it look like?

When any user is logged in, they'll see the *My Borrowed* link in the sidebar, and the list of books displayed as below (the first book has no due date, which is a bug we hope to fix in a later tutorial!).

[Home](#)
[All books](#)
[All authors](#)

User: superman
[My Borrowed](#)
[Logout](#)

Borrowed books

- [The Wise Man's Fear](#) (None)
- [The Name of the Wind](#) (Sept. 6, 2016)

[Permissions](#)[Edit](#)

Permissions are associated with models, and define the operations that can be performed on a model instance by a user who has the permission. By default, Django automatically gives *add*, *change*, and *delete* permissions to all models, which allow users with the permissions to perform the associated actions via the admin site. You can define your own permissions to models and grant them to specific users. You can also change the permissions associated with different instances of the same model.

Testing on permissions in views and templates is then very similar for testing on the authentication status (and in fact, testing for a permission also tests for authentication).

[Models](#)

Defining permissions is done on the model "class Meta" section, using the `permissions` field. You can specify as many permissions as you need in a tuple, each permission itself being defined in a nested tuple containing the permission name and permission display value. For example, we might define a permission to allow a user to mark that a book has been returned as shown:

```
class BookInstance(models.Model):
    ...
    class Meta:
        ...
        permissions = (("can_mark_returned", "Set book as returned"),)
```

We could then assign the permission to a "Librarian" group in the Admin site.

Open the **catalog/models.py**, and add the permission as shown above. You will need to re-run your migrations (call `python3 manage.py makemigrations` and `python3 manage.py migrate`) to update the database appropriately.

[Templates](#)

The current user's permissions are stored in a template variable called `{{ perms }}`. You can check whether the current user has a particular permission using the specific variable name within the associated Django "app" — e.g. `{{ perms.catalog.can_mark_returned }}` will be `True` if the user has this permission, and `False` otherwise. We typically test for the permission using the template `{% if %}` tag as shown:

```
{% if perms.catalog.can_mark_returned %}
    <!-- We can mark a BookInstance as returned. -->
    <!-- Perhaps add code to link to a "book return" view here. -->
{% endif %}
```

Views

Permissions can be tested in function view using the `permission_required` decorator or in a class-based view using the `PermissionRequiredMixin`. The pattern and behaviour are the same as for login authentication, though of course you might reasonably have to add multiple permissions.

Function view decorator:

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

@permission_required('catalog.can_mark_returned')
@permission_required('catalog.can_edit')
def my_view(request):
    ...
```

Permission-required mixin for class-based views.

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

class MyView(PermissionRequiredMixin, View):
    permission_required = 'catalog.can_mark_returned'
    # Or multiple of permissions:
    permission_required = ('catalog.can_mark_returned', 'catalog.can_edit')
```

Example

We won't update the *LocalLibrary* here; perhaps in the next tutorial!

Challenge yourself [Edit](#)

Earlier in this article we showed you how to create a page for the current user listing the books that they have borrowed. The challenge now is to create a similar page that is only visible for librarians, that displays *all* books that have been borrowed, and which includes the name of each borrower.

You should be able to follow the same pattern as for the other view. The main difference is that you'll need to restrict the view to only librarians. You could do this based on whether the user is a staff member (function decorator: `staff_member_required`, template variable: `user.is_staff`) but we recommend that you instead use the `can_mark_returned` permission and `PermissionRequiredMixin`, as described in the previous section.

Important: Remember not to use your superuser for permissions based testing (permission checks always return true for superusers, even if a permission has not yet been defined!). Instead create a librarian user, and add the required capability.

When you are finished, your page should look something like the screenshot below.

[Home](#)
[All books](#)
[All authors](#)

User: superman
[My Borrowed](#)
[Logout](#)

[Staff](#)
[All borrowed](#)

All Borrowed Books

- [The Name of the Wind](#) (Sept. 6, 2016) - superman
- [The Dueling Machine](#) (Sept. 18, 2016) - johnc
- [The Wise Man's Fear](#) (Oct. 12, 2016) - superman

[Summary](#)[Edit](#)

Excellent work — you've now created a website that library members can login into and view their own content, and that librarians (with the correct permission) can use to view all loaned books and their borrowers. At the moment we're still just viewing content, but the same principles and techniques are used when you want to start modifying and adding data.

In our next article we'll look at how you can use Django forms to collect user input, and then start modifying some of our stored data.

Django Tutorial Part 9: Working with forms

In this tutorial we'll show you how to work with HTML Forms in Django, and in particular the easiest way to write forms to create, update, and delete model instances. As part of this demonstration we'll extend the [LocalLibrary](#) website so that librarians can renew books, and create, update, and delete authors using our own forms (rather than using the admin application).

Prerequisites: Complete all previous tutorial topics, including [Django Tutorial Part 8: User authentication and permissions](#).






Objective: To understand how write forms to get information from users and update the database. To understand how the generic class-based form editing views can vastly simplify creating forms for working with a single model.

[Overview](#)[Edit](#)

An [HTML Form](#) is a group of one or more fields/widgets on a web page, which can be used to collect information from users for submission to a server. Forms are a flexible mechanism for collecting user input because there are suitable widgets for entering many different types of data, including text boxes, checkboxes, radio buttons, date pickers, etc. Forms are also a relatively secure way of sharing data with the server, as they allow us to send data in `POST` requests with cross-site request forgery protection.

While we haven't created any forms in this tutorial so far, we've already encountered them in the Django Admin site — for example the screenshot below shows a form for editing a one of our [Book](#) models, comprised of a number of selection lists and text editors.

Add book

Title:	<input type="text"/>
Author:	<input type="text" value="-----"/>    
Summary:	<div><div></div><div>Enter a brief description of the book</div></div>
ISBN:	<input type="text"/> 13 Character ISBN number
Genre:	<div><div>Science Fiction Fantasy Western French Poetry</div><div></div></div> <p>Select a genre for this book Hold down "Control", or "Command" on a Mac, to select more than one.</p>
<div><div>Save and add another</div><div>Save and continue editing</div><div>SAVE</div></div>	

Working with forms can be complicated! Developers need to write HTML for the form, validate and properly sanitise entered data on the server (and possibly also in the browser), repost the form with error messages to inform users of any invalid fields, handle the data when it has successfully been submitted, and finally respond to the user in some way to indicate success. Django Forms take a lot of the work out of all these steps, by providing a framework that lets you define forms and their fields programmatically, and then use these objects to both generate the form HTML code and handle much of the validation and user interaction.

In this tutorial we're going to show you a few of the ways you can create and work with forms, and in particular, how the generic editing form views can significantly reduce the amount of work you need to do to create forms to manipulate your models. Along the way we'll extend our *LocalLibrary* application by adding a form to allow librarians to renew library books, and we'll create pages to create, edit and delete books and authors (reproducing a basic version of the form shown above for editing books).

First a brief overview of [HTML Forms](#). Consider a simple HTML form, with a single text field for entering the name of some "team", and its associated label:

Enter name:

The form is defined in HTML as a collection of elements inside `<form>...</form>` tags, containing at least one `input` element of `type="submit"`.

```
<form action="/team_name_url/" method="post">
  <label for="team_name">Enter name: </label>
  <input id="team_name" type="text" name="name_field" value="Default name
for team.">
  <input type="submit" value="OK">
</form>
```

While here we just have one text field for entering the team name, a form *may* have any number of other input elements and their associated labels. The field's `type` attribute defines what sort of widget will be displayed. The `name` and `id` of the field are used to identify the field in JavaScript/CSS/HTML, while `value` defines the initial value for the field when it is first displayed. The matching team label is specified using the `label` tag (see "Enter name" above), with a `for` field containing the `id` value of the associated `input`.

The `submit` input will be displayed as a button (by default) that can be pressed by the user to upload the data in all the other input elements in the form to the server (in this case, just the `team_name`). The form attributes define the HTTP `method` used to send the data and the destination of the data on the server (`action`):

- `action`: The resource/URL where data is to be sent for processing when the form is submitted. If this is not set (or set to an empty string), then the form will be submitted back to the current page URL.
- `method`: The HTTP method used to send the data: *post* or *get*.
 - The `POST` method should always be used if the data is going to result in a change to the server's database, because this can be made more resistant to cross-site forgery request attacks.
 - The `GET` method should only be used for forms that don't change user data (e.g. a search form). It is recommended for when you want to be able to bookmark or share the URL.

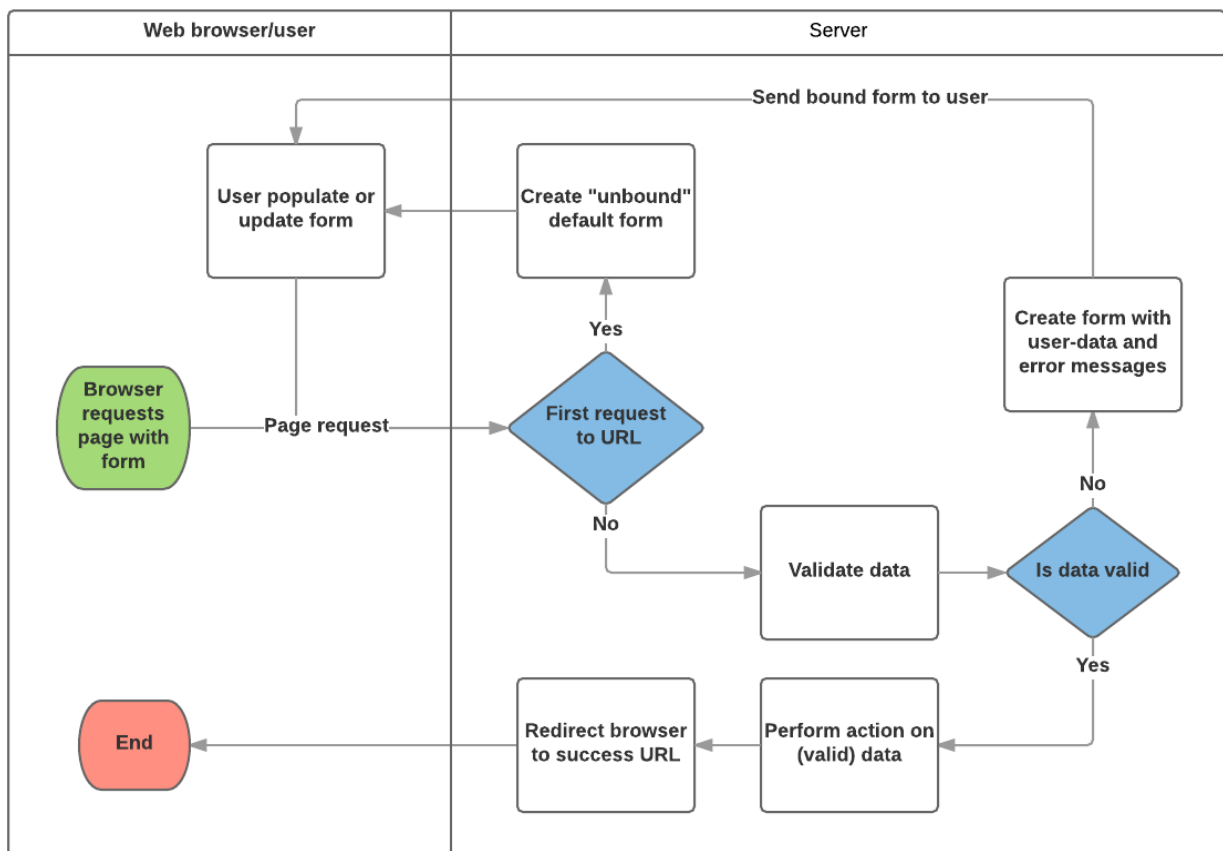
The role of the server is first to render the initial form state — either containing blank fields, or pre-populated with initial values. After the user presses the submit button the server will receive the form data with values from the web browser, and must validate the information. If the form contains invalid data the server should display the form again, this time with user-entered data in "valid" fields, and messages to describe the problem for the invalid fields. Once the server gets a request with all valid form data it can perform an appropriate action (e.g. saving the data, returning the result of a search, uploading a file etc.) and then notify the user.

As you can imagine, creating the HTML, validating the returned data, re-displaying the entered data with error reports if needed, and performing the desired operation on valid data can all take quite a lot of effort to "get right". Django makes this a lot easier, by taking away some of the heavy lifting and repetitive code!

Django form handling process [Edit](#)

Django's form handling uses all of the same techniques that we learned about in previous tutorials (for displaying information about our models): the view gets a request, performs any actions required including reading data from the models, then generates and returns an HTML page (from a template, into which we pass a *context* containing the data to be displayed). What makes things more complicated is that the server also needs to be able to process data provided by the user, and redisplay the page if there are any errors.

A process flowchart of how Django handles form requests is shown below, starting with a request for a page containing a form (shown in green).



Based on the diagram above, the main things that Django's form handling does are:

1. Display the default form the first time it is requested by the user.
 - The form may contain blank fields (e.g. if you're creating a new record), or it may be pre-populated with initial values (e.g. if you are changing a record, or have useful default initial values).

- The form is referred to as *unbound* at this point, because it isn't associated with any user-entered data (though it may have initial values).
- 2. Receive data from a submit request and bind it to the form.
 - Binding data to the form means that the user-entered data and any errors are available when we need to redisplay the form.
- 3. Clean and validate the data.
 - Cleaning the data performs sanitisation of the input (e.g. removing invalid characters that might potentially be used to send malicious content to the server) and converts them into consistent Python types.
 - Validation checks that the values are appropriate for the field (e.g. are in the right date range, aren't too short or too long, etc.)
- 4. If any data is invalid, re-display the form, this time with any user populated values and error messages for the problem fields.
- 5. If all data is valid, perform required actions (e.g. save the data, send an email, return the result of a search, upload a file etc.)
- 6. Once all actions are complete, redirect the user to another page.

Django provides a number of tools and approaches to help you with the tasks detailed above. The most fundamental is the `Form` class, which simplifies both generation of form HTML and data cleaning/validation. In the next section we describe how forms work using the practical example of a page to allow librarians to renew books.

Note: Understanding how `Form` is used will help you when we discuss Django's more "high level" form framework classes.

[Renew-book form using a Form and function view](#)[Edit](#)

Next we're going to add a page to allow librarians to renew borrowed books. To do this we'll create a form that allows users to enter a date value. We'll seed the field with an initial value 3 weeks from the current date (the normal borrowing period), and add some validation to ensure that the librarian can't enter a date in the past or a date too far in the future. When a valid date has been entered, we'll write it to the current record's `BookInstance.due_back` field.

The example will use a function-based view and a `Form` class. The following sections explain how forms work, and the changes you need to make to our ongoing *LocalLibrary* project.

Form

The `Form` class is the heart of Django's form handling system. It specifies the fields in the form, their layout, display widgets, labels, initial values, valid values, and (once validated) the error messages associated with invalid fields. The class also provides methods for rendering itself in templates using predefined formats (tables, lists, etc.) or for getting the value of any element (enabling fine-grained manual rendering).

[Declaring a Form](#)

The declaration syntax for a `Form` is very similar to that for declaring a `Model`, and shares the same field types (and some similar parameters). This makes sense because in both cases we need to ensure that each field handles the right types of data, is constrained to valid data, and has a description for display/documentation.

To create a `Form` we import the `forms` library, derive from the `Form` class, and declare the form's fields. A very basic form class for our library book renewal form is shown below:

```
from django import forms

class RenewBookForm(forms.Form):
    renewal_date = forms.DateField(help_text="Enter a date between now and 4 weeks (default 3).")
```

Form fields

In this case we have a single [DateField](#) for entering the renewal date that will render in HTML with a blank value, the default label "*Renewal date:*", and some helpful usage text: "*Enter a date between now and 4 weeks (default 3 weeks).*" As none of the other optional arguments are specified the field will accept dates using the [input formats](#): YYYY-MM-DD (2016-11-06), MM/DD/YYYY (02/26/2016), MM/DD/YY (10/25/16), and will be rendered using the default [widget](#): [DateInput](#).

There are many other types of form fields, which you will largely recognise from their similarity to the equivalent model field classes: [BooleanField](#), [CharField](#), [ChoiceField](#), [TypedChoiceField](#), [DateField](#), [DateTimeField](#), [DecimalField](#), [DurationField](#), [EmailField](#), [FileField](#), [FilePathField](#), [FloatField](#), [ImageField](#), [IntegerField](#), [GenericIPAddressField](#), [MultipleChoiceField](#), [TypedMultipleChoiceField](#), [NullBooleanField](#), [RegexField](#), [SlugField](#), [TimeField](#), [URLField](#), [UUIDField](#), [ComboField](#), [MultiValueField](#), [SplitDateTimeField](#), [ModelMultipleChoiceField](#), [ModelChoiceField](#).

The arguments that are common to most fields are listed below (these have sensible default values):

- [required](#): If `True`, the field may not be left blank or given a `None` value. Fields are required by default, so you would set `required=False` to allow blank values in the form.
- [label](#): The label to use when rendering the field in HTML. If [label](#) is not specified then Django would create one from the field name by capitalising the first letter and replacing underscores with spaces (e.g. *Renewal date*).
- [label_suffix](#): By default a colon is displayed after the label (e.g. *Renewal date:*). This argument allows you to specify as different suffix containing other character(s).
- [initial](#): The initial value for the field when the form is displayed.
- [widget](#): The display widget to use.
- [help_text](#) (as seen in the example above): Additional text that can be displayed in forms to explain how to use the field.
- [error_messages](#): A list of error messages for the field. You can override these with your own messages if needed.
- [validators](#): A list of functions that will be called on the field when it is validated.
- [localize](#): Enables the localisation of form data input (see link for more information).
- [disabled](#): The field is displayed but its value cannot be edited if this is `True`. The default is `False`.

Validation

Django provides numerous places where you can validate your data. The easiest way to validate a single field is to override the method `clean_<fieldname>()` for the field you want to check.

So for example, we can validate that entered `renewal_date` values are between now and 4 weeks by implementing `clean_renewal_date()` as shown below.

```
from django import forms

from django.core.exceptions import ValidationError
from django.utils.translation import ugettext_lazy as _
import datetime #for checking renewal date range.

class RenewBookForm(forms.Form):
    renewal_date = forms.DateField(help_text="Enter a date between now and 4 weeks (default 3).")

    def clean_renewal_date(self):
        data = self.cleaned_data['renewal_date']

        #Check date is not in past.
        if data < datetime.date.today():
            raise ValidationError(_('Invalid date - renewal in past'))

        #Check date is in range librarian allowed to change (+4 weeks).
        if data > datetime.date.today() + datetime.timedelta(weeks=4):
            raise ValidationError(_('Invalid date - renewal more than 4 weeks ahead'))

        # Remember to always return the cleaned data.
        return data
```

There are two important things to note. The first is that we get our data using `self.cleaned_data['renewal_date']` and that we return this data whether or not we change it at the end of the function. This step gets us the data "cleaned" and sanitised of potentially unsafe input using the default validators, and converted into the correct standard type for the data (in this case a Python `datetime.datetime` object).

The second point is that if a value falls outside our range we raise a `ValidationError`, specifying the error text that we want to display in the form if an invalid value is entered. The example above also wraps this text in one of [Django's translation functions](#) `ugettext_lazy()` (imported as `_()`), which is good practice if you want to translate your site later.

Note: There are numerous other methods and examples for validating forms in [Form and field validation](#) (Django docs). For example, in cases where you have multiple fields that depend on each other, you can override the [Form.clean\(\)](#) function and again raise a `ValidationError`.

That's all we need for the form in this example!

Copy the Form

Create and open the file **locallibrary/catalog/forms.py** and copy the entire code listing from the previous block into it.

URL Configuration

Before we create our view, let's add a URL configuration for the *renew-books* page. Copy the following configuration to the bottom of **locallibrary/catalog/urls.py**.

```
urlpatterns += [
    url(r'^book/(?P<pk>[-\w]+)/renew/$', views.renew_book_librarian,
        name='renew-book-librarian'),
]
```

The URL configuration will redirect URLs with the format `/catalog/book/<bookinstance id>/renew/` to the function named `renew_book_librarian()` in **views.py**, and send the `BookInstance id` as the parameter named `pk`.

Note: We can name our captured URL data "pk" anything we like, because we have complete control over the view function (we're not using a generic detail view class that expects parameters with a certain name). However `pk`, short for "primary key", is a reasonable convention to use!

View

As discussed in the [Django form handling process](#) above, the view has to render the default form when it is first called and then either re-render it with error messages if the data is invalid, or process the data and redirect to a new page if the data is valid. In order to perform these different actions, the view has to be able to know whether it is being called for the first time to render the default form, or a subsequent time to validate data.

For forms that use a `POST` request to submit information to the server, the most common pattern is for the view to test against the `POST` request type (if `request.method == 'POST':`) to identify form validation requests and `GET` (using an `else` condition) to identify the initial form creation request. If you want to submit your data using a `GET` request then a typical approach for identifying whether this is the first or subsequent view invocation is to read the form data (e.g. to read a hidden value in the form).

The book renewal process will be writing to our database, so by convention we use the `POST` request approach. The code fragment below shows the (very standard) pattern for this sort of function view.

```
from django.shortcuts import get_object_or_404
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
import datetime

from .forms import RenewBookForm

def renew_book_librarian(request, pk):
    book_inst=get_object_or_404(BookInstance, pk = pk)

    # If this is a POST request then process the Form data
    if request.method == 'POST':

        # Create a form instance and populate it with data from the request
        (binding):
        form = RenewBookForm(request.POST)

        # Check if the form is valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required (here we just
            write it to the model due_back field)
```



```

        book_inst.due_back = form.cleaned_data['renewal_date']
        book_inst.save()

        # redirect to a new URL:
        return HttpResponseRedirect(reverse('all-borrowed') )

    # If this is a GET (or any other method) create the default form.
    else:
        proposed_renewal_date = datetime.date.today() +
datetime.timedelta(weeks=3)
        form = RenewBookForm(initial={'renewal_date':
proposed_renewal_date,})

        return render(request, 'catalog/book_renew_librarian.html', {'form':
form, 'bookinst':book_inst})

```

First we import our form (`RenewBookForm`) and a number of other useful objects/methods used in the body of the view function:

- [`get_object_or_404\(\)`](#): Returns a specified object from a model based on its primary key value, and raises an `Http404` exception (not found) if the record does not exist.
- [`HttpResponseRedirect`](#): This creates a redirect to a specified URL (HTTP status code 302).
- [`reverse\(\)`](#): This generates a URL from a URL configuration name and a set of arguments. It is the Python equivalent of the `url` tag that we've been using in our templates.
- [`datetime`](#): A Python library for manipulating dates and times.

In the view we first use the `pk` argument in `get_object_or_404()` to get the current `BookInstance` (if this does not exist, the view will immediately exit and the page will display a "not found" error). If this is *not* a POST request (handled by the `else` clause) then we create the default form passing in an initial value for the `renewal_date` field (as shown in bold below, this is 3 weeks from the current date).

```

book_inst=get_object_or_404(BookInstance, pk = pk)

    # If this is a GET (or any other method) create the default form
    else:
        proposed_renewal_date = datetime.date.today() +
datetime.timedelta(weeks=3)
        form = RenewBookForm(initial={'renewal_date':
proposed_renewal_date,})

        return render(request, 'catalog/book_renew_librarian.html', {'form':
form, 'bookinst':book_inst})

```

After creating the form, we call `render()` to create the HTML page, specifying the template and a context that contains our form. In this case the context also contains our `BookInstance`, which we'll use in the template to provide information about the book we're renewing.

If however this is a POST request, then we create our `form` object and populate it with data from the request. This process is called "binding" and allows us to validate the form. We then check if the form is valid, which runs all the validation code on all of the fields — including both the generic code to check that our date field is actually a valid date and our specific form's `clean_renewal_date()` function to check the date is in the right range.

```

book_inst=get_object_or_404(BookInstance, pk = pk)

```

```

# If this is a POST request then process the Form data
if request.method == 'POST':

    # Create a form instance and populate it with data from the request
    (binding):
    form = RenewBookForm(request.POST)

    # Check if the form is valid:
    if form.is_valid():
        # process the data in form.cleaned_data as required (here we just
        write it to the model due_back field)
        book_inst.due_back = form.cleaned_data['renewal_date']
        book_inst.save()

        # redirect to a new URL:
        return HttpResponseRedirect(reverse('all-borrowed') )

    return render(request, 'catalog/book_renew_librarian.html', {'form':
form, 'bookinst':book_inst})

```

If the form is not valid we call `render()` again, but this time the form value passed in the context will include error messages.

If the form is valid, then we can start to use the data, accessing it through the `form.cleaned_data` attribute (e.g. `data = form.cleaned_data['renewal_date']`). Here we just save the data into the `due_back` value of the associated `BookInstance` object.

Important: While you can also access the form data directly through the request (for example `request.POST['renewal_date']` or `request.GET['renewal_date']` (if using a GET request) this is NOT recommended. The cleaned data is sanitised, validated, and converted into Python-friendly types.

The final step in the form-handling part of the view is to redirect to another page, usually a "success" page. In this case we use `HttpResponseRedirect` and `reverse()` to redirect to the view named 'all-borrowed' (this was created as the "challenge" in [Django Tutorial Part 8: User authentication and permissions](#)). If you didn't create that page consider redirecting to the home page at URL '/').

That's everything needed for the form handling itself, but we still need to restrict access to the view to librarians. We should probably create a new permission in `BookInstance` ("can_renew"), but to keep things simple here we just use the `@permission_required` function decorator with our existing `can_mark_returned` permission.

The final view is therefore as shown below. Please copy this into the bottom of `locallibrary/catalog/views.py`.

```

from django.contrib.auth.decorators import permission_required

from django.shortcuts import get_object_or_404
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
import datetime

from .forms import RenewBookForm

```

```

@permission_required('catalog.can_mark_returned')
def renew_book_librarian(request, pk):
    """
    View function for renewing a specific BookInstance by librarian
    """
    book_inst=get_object_or_404(BookInstance, pk = pk)

    # If this is a POST request then process the Form data
    if request.method == 'POST':

        # Create a form instance and populate it with data from the request
        (binding):
        form = RenewBookForm(request.POST)

        # Check if the form is valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required (here we just
            write it to the model due_back field)
            book_inst.due_back = form.cleaned_data['renewal_date']
            book_inst.save()

            # redirect to a new URL:
            return HttpResponseRedirect(reverse('all-borrowed') )

        # If this is a GET (or any other method) create the default form.
        else:
            proposed_renewal_date = datetime.date.today() +
            datetime.timedelta(weeks=3)
            form = RenewBookForm(initial={'renewal_date':
            proposed_renewal_date,})

            return render(request, 'catalog/book_renew_librarian.html', {'form':
            form, 'bookinst':book_inst})

```

The template

Create the template referenced in the view

(/catalog/templates/catalog/book_renew_librarian.html) and copy the code below into it:

```

{% extends "base_generic.html" %}
{% block content %}

    <h1>Renew: {{bookinst.book.title}}</h1>
    <p>Borrower: {{bookinst.borrower}}</p>
    <p{% if bookinst.is_overdue %} class="text-danger"{% endif %}>Due date:
    {{bookinst.due_back}}</p>

    <form action="" method="post">
        {% csrf_token %}
        <table>
            {{ form }}
        </table>
        <input type="submit" value="Submit" />
    </form>

{% endblock %}

```

Most of this will be completely familiar from previous tutorials. We extend the base template and then redefine the content block. We are able to reference {{bookinst}} (and its variables)

because it was passed into the context object in the `render()` function, and we use these to list the book title, borrower and the original due date.

The form code is relatively simple. First we declare the `form` tags, specifying where the form is to be submitted (`action`) and the method for submitting the data (in this case an "HTTP POST") — if you recall the [HTML Forms](#) overview at the top of the page, an empty `action` as shown, means that the form data will be posted back to the current URL of the page (which is what we want!). Inside the tags we define the `submit` input, which a user can press to submit the data. The `{% csrf_token %}` added just inside the form tags is part of Django's cross-site forgery protection.

Note: Add the `{% csrf_token %}` to every Django template you create that uses `POST` to submit data. This will reduce the chance of forms being hijacked by malicious users.

All that's left is the `{{ form }}` template variable, which we passed to the template in the context dictionary. Perhaps unsurprisingly, when used as shown this provides the default rendering of all the form fields, including their labels, widgets, and help text — the rendering is as shown below:

```
<tr>
  <th><label for="id_renewal_date">Renewal date:</label></th>
  <td>
    <input id="id_renewal_date" name="renewal_date" type="text" value="2016-11-08" required />
    <br />
    <span class="helptext">Enter date between now and 4 weeks (default 3 weeks).</span>
  </td>
</tr>
```

Note: It is perhaps not obvious because we only have one field, but by default every field is defined in its own table row (which is why the variable is inside `table` tags above). This same rendering is provided if you reference the template variable `{{ form.as_table }}`.

If you were to enter an invalid date, you'd additionally get a list of the errors rendered in the page (shown in bold below).

```
<tr>
  <th><label for="id_renewal_date">Renewal date:</label></th>
  <td>
    <ul class="errorlist">
      <li>Invalid date - renewal in past</li>
    </ul>
    <input id="id_renewal_date" name="renewal_date" type="text" value="2015-11-08" required />
    <br />
    <span class="helptext">Enter date between now and 4 weeks (default 3 weeks).</span>
  </td>
</tr>
```

Other ways of using form template variable

Using `{{ form }}` as shown above, each field is rendered as a table row. You can also render each field as a list item (using `{{ form.as_ul }}`) or as a paragraph (using `{{ form.as_p }}`).

What is even more cool is that you can have complete control over the rendering of each part of the form, by indexing its properties using dot notation. So for example we can access a number of separate items for our `renewal_date` field:

- `{{form.renewal_date}}`: The whole field.
- `{{form.renewal_date.errors}}`: The list of errors.
- `{{form.renewal_date.id_for_label}}`: The id of the label.
- `{{form.renewal_date.help_text}}`: The field help text.
- etc!

For more examples of how to manually render forms in templates and dynamically loop over template fields, see [Working with forms > Rendering fields manually](#) (Django docs).

Testing the page

If you accepted the "challenge" in [Django Tutorial Part 8: User authentication and permissions](#) you'll have a list of all books on loan in the library, which is only visible to library staff. We can add a link to our renew page next to each item using the template code below.

```
{% if perms.catalog.can_mark_returned %}- <a href="{% url 'renew-book-librarian' bookinst.id %}">Renew</a> {% endif %}
```

Note: Remember that your test login will need to have the permission "catalog.can_mark_returned" in order to access the renew book page (perhaps use your superuser account).

You can alternatively manually construct a test URL like this — http://127.0.0.1:8000/catalog/book/<bookinstance_id>/renew/ (a valid bookinstance id can be obtained by navigating to a book detail page in your library, and copying the `id` field).

What does it look like?

If you are successful, the default form will look like this:

← → ↻ ⓘ 127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Renew: The Wise Man's Fear

Borrower: superman

Due date: Oct. 12, 2016

Renewal date:

Enter date between now and 4 weeks (default 3 weeks).

The form with an invalid value entered, will look like this:

← → ↻ ⓘ 127.0.0.1:8000/catalog/book/d2ad0f63-82b0-46ac-8511-c89b76756a6b/renew/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Renew: The Wise Man's Fear

Borrower: superman

Due date: Oct. 12, 2016

- Invalid date - renewal in past

Renewal date:

Enter date between now and 4 weeks (default 3 weeks).

Submit

The list of all books with renew links will look like this:

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

All Borrowed Books

- The Wise Man's Fear (Oct. 12, 2016) - superman - [Renew](#)
- The Name of the Wind (Nov. 8, 2016) - superman - [Renew](#)
- The Dueling Machine (Nov. 8, 2016) - johnc - [Renew](#)

[ModelFormsEdit](#)

Creating a `Form` class using the approach described above is very flexible, allowing you to create whatever sort of form page you like and associate it with any model or models.

However if you just need a form to map the fields of a *single* model then your model will already define most of the information that you need in your form: fields, labels, help text, etc. Rather than recreating the model definitions in your form, it is easier to use the [ModelForm](#) helper class to create the form from your model. This `ModelForm` can then be used within your views in exactly the same way as an ordinary `Form`.

A basic `ModelForm` containing the same field as our original `RenewBookForm` is shown below. All you need to do to create the form is add `class Meta` with the associated `model` (`BookInstance`) and a list of the model `fields` to include in the form (you can include all fields

using `fields = '__all__'`, or you can use `exclude` (instead of `fields`) to specify the fields *not* to include from the model).

```
from django.forms import ModelForm
from .models import BookInstance
```

```
class RenewBookModelForm(ModelForm):
    class Meta:
        model = BookInstance
        fields = ['due_back',]
```

Note: This might not look like all that much simpler than just using a `Form` (and it isn't in this case, because we just have one field). However if you have a lot of fields, it can reduce the amount of code quite significantly!

The rest of the information comes from the model field definitions (e.g. labels, widgets, help text, error messages). If these aren't quite right, then we can override them in our `class Meta`, specifying a dictionary containing the field to change and its new value. For example, in this form we might want a label for our field of "*Renewal date*" (rather than the default based on the field name: *Due date*), and we also want our help text to be specific to this use case. The `Meta` below shows you how to override these fields, and you can similarly set `widgets` and `error_messages` if the defaults aren't sufficient.

```
class Meta:
    model = BookInstance
    fields = ['due_back',]
    labels = { 'due_back': _('Renewal date'), }
    help_texts = { 'due_back': _('Enter a date between now and 4 weeks
(default 3).'), }
```

To add validation you can use the same approach as for a normal `Form` — you define a function named `clean_field_name()` and raise `ValidationError` exceptions for invalid values. The only difference with respect to our original form is that the model field is named `due_back` and not "`renewal_date`".

```
from django.forms import ModelForm
from .models import BookInstance
```

```
class RenewBookModelForm(ModelForm):
    def clean_due_back(self):
        data = self.cleaned_data['due_back']

        #Check date is not in past.
        if data < datetime.date.today():
            raise ValidationError(_('Invalid date - renewal in past'))

        #Check date is in range librarian allowed to change (+4 weeks)
        if data > datetime.date.today() + datetime.timedelta(weeks=4):
            raise ValidationError(_('Invalid date - renewal more than 4 weeks
ahead'))

        # Remember to always return the cleaned data.
        return data

    class Meta:
        model = BookInstance
```

```

        fields = ['due_back',]
        labels = { 'due_back': _('Renewal date'), }
        help_texts = { 'due_back': _('Enter a date between now and 4 weeks
(default 3).'), }

```

The class `RenewBookModelForm` below is now functionally equivalent to our original `RenewBookForm`. You could import and use it wherever you currently use `RenewBookForm`.

Generic editing views [Edit](#)

The form handling algorithm we used in our function view example above represents an extremely common pattern in form editing views. Django abstracts much of this "boilerplate" for you, by creating [generic editing views](#) for creating, editing, and deleting views based on models. Not only do these handle the "view" behaviour, but they automatically create the form class (a `ModelForm`) for you from the model.

Note: In addition to the editing views described here, there is also a [FormView](#) class, which lies somewhere between our function view and the other generic views in terms of "flexibility" vs "coding effort". Using `FormView` you still need to create your `Form`, but you don't have to implement all of the standard form-handling pattern. Instead you just have to provide an implementation of the function that will be called once the submitted is known to be valid.

In this section we're going to use generic editing views to create pages to add functionality to create, edit, and delete `Author` records from our library — effectively providing a basic reimplementaion of parts of the Admin site (this could be useful if you need to offer admin functionality in a more flexible way that can be provided by the admin site).

Views

Open the views file (**locallibrary/catalog/views.py**) and append the following code block to the bottom of it:

```

from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy
from .models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = '__all__'
    initial={'date_of_death':'12/10/2016',}

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['first_name', 'last_name', 'date_of_birth', 'date_of_death']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('authors')

```

As you can see, to create the views you need to derive from `CreateView`, `UpdateView`, and `DeleteView` (respectively) and then define the associated model.

For the "create" and "update" cases you also need to specify the fields to display in the form (using in same syntax as for `ModelForm`). In this case we show both the syntax to display "all" fields, and how you can list them individually. You can also specify initial values for each of the fields using a dictionary of *field_name/value* pairs (here we arbitrarily set the date of death for demonstration purposes — you might want to remove that!). By default these views will redirect on success to a page displaying the newly created/edited model item, which in our case will be the author detail view we created in a previous tutorial. You can specify an alternative redirect location by explicitly declaring parameter `success_url` (as done for the `AuthorDelete` class).

The `AuthorDelete` class doesn't need to display any of the fields, so these don't need to be specified. You do however need to specify the `success_url`, because there is no obvious default value for Django to use. In this case we use the [`reverse_lazy\(\)`](#) function to redirect to our author list after an author has been deleted — `reverse_lazy()` is a lazily executed version of `reverse()`, used here because we're providing an URL to a class-based view attribute.

Templates

The "create" and "update" views use the same template by default, which will be named after your model: **`model_name_form.html`** (you can change the suffix to something other than **`_form`** using the `template_name_suffix` field in your view, e.g. `template_name_suffix = '_other_suffix'`)

Create the template file **`locallibrary/catalog/templates/catalog/author_form.html`** and copy in the text below.

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

{% block content %}

<form action="" method="post">
    {% csrf_token %}
    <table>
    {{ form.as_table }}
    </table>
    <input type="submit" value="Submit" />

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

This is similar to our previous forms, and renders the fields using a table. Note also how again we declare the `{% csrf_token %}` to ensure that our forms are resistant to CSRF attacks.

The "delete" view expects to find a template named with the format **`model_name_confirm_delete.html`** (again, you can change the suffix using `template_name_suffix` in your view). Create the template file **`locallibrary/catalog/templates/catalog/author_confirm_delete.html`** and copy in the text below.

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

{% block content %}

<h1>Delete Author</h1>
```

```
<p>Are you sure you want to delete the author: {{ author }}?</p>

<form action="" method="POST">
    {% csrf_token %}
    <input type="submit" action="" value="Yes, delete." />
</form>

{% endblock %}
URL configurations
```

Open your URL configuration file (**locallibrary/catalog/urls.py**) and add the following configuration to the bottom of the file:

```
urlpatterns += [
    url(r'^author/create/$', views.AuthorCreate.as_view(),
        name='author_create'),
    url(r'^author/(?P<pk>\d+)/update/$', views.AuthorUpdate.as_view(),
        name='author_update'),
    url(r'^author/(?P<pk>\d+)/delete/$', views.AuthorDelete.as_view(),
        name='author_delete'),
]
```

There is nothing particularly new here! You can see that the views are classes, and must hence be called via `.as_view()`, and you should be able to recognise the URL patterns in each case. We must use `pk` as the name for our captured primary key value, as this is the parameter name expected by the view classes.

The author create, update, and delete pages are now ready to test (we won't bother hooking them into the site sidebar in this case, although you can do so if you wish).

Note: Observant users will have noticed that we didn't do anything to prevent unauthorised users from accessing the pages! We leave that as an exercise for you (hint: you could use the `PermissionRequiredMixin` and either create a new permission or reuse our `can_mark_returned` permission).

Testing the page

First login to the site with an account that has whatever permissions you decided are needed to access the author editing pages.

Then navigate to the author create page: <http://127.0.0.1:8000/catalog/author/create/>, which should look like the screenshot below.

← → ↻ ⓘ 127.0.0.1:8000/catalog/author/create/

Home
All books
All authors
User: superman
My Borrowed

First name:
Last name:
Date of birth:
Died: 12/10/2016

Enter values for the fields and then press **Submit** to save the author record. You should now be taken to a detail view for your new author, with an URL of something like *http://127.0.0.1:8000/catalog/author/10*.

You can test editing records by appending */update/* to the end of the detail view URL (e.g. *http://127.0.0.1:8000/catalog/author/10/update/*) — we don't show a screenshot, because it looks just like the "create" page!

Last of all we can delete the page, by appending *delete* to the end of the author detail-view URL (e.g. *http://127.0.0.1:8000/catalog/author/10/delete/*). Django should display the delete page shown below. Press **Yes, delete.** to remove the record and be taken to the list of all authors.

← → ↻ ⓘ 127.0.0.1:8000/catalog/author/10/delete/

Home
All books
All authors
User: superman
My Borrowed
Logout

Delete Author

Are you sure you want to delete the author: Twain, Mark?

Challenge yourself [Edit](#)

Create some forms to create, edit and delete `Book` records. You can use exactly the same structure as for `Authors`. If your **book_form.html** template is just a copy-renamed version of the **author_form.html** template, then the new "create book" page will look like the screenshot below:

← → ↻ ⓘ 127.0.0.1:8000/catalog/book/create/

Home
All books
All authors

User: superman
My Borrowed
Logout

Staff
All borrowed

Title:

Author:

Summary:

Enter a brief description of the book

ISBN:

13 Character ISBN number

Genre:

Science Fiction
Fantasy
Western
French Poetry

Select a genre for this book

Language:

Submit

[Summary](#)[Edit](#)

Creating and handling forms can be a complicated process! Django makes it much easier by providing programmatic mechanisms to declare, render and validate forms. Furthermore, Django provides generic form editing views that can do *almost all* the work to define pages that can create, edit, and delete records associated with a single model instance.

There is a lot more that can be done with forms (check out our See also list below), but you should now understand how to add basic forms and form-handling code to your own websites.

Django Tutorial Part 10: Testing a Django web application

As websites grow they become harder to test manually. Not only is there more to test, but, as interactions between components become more complex, a small change in one area can impact other areas, so more changes will be required to ensure everything keeps working and errors are not introduced as more changes are made. One way to mitigate these problems is to write automated tests, which can easily and reliably be run every time you make a change. This tutorial shows how to automate *unit testing* of your website using Django's test framework.

Prerequisites: Complete all previous tutorial topics, including [Django Tutorial Part 9: Working with forms](#).

Objective: To understand how to write unit tests for Django-based websites.

[Overview](#)[Edit](#)

The [LocalLibrary](#) currently has pages to display lists of all books and authors, detail views for `Book` and `Author` items, a page to renew `BookInstances`, and pages to create, update, and delete `Author` items (and `Book` records too, if you completed the *challenge* in the [forms tutorial](#)). Even with this relatively small site, manually navigating to each page and *superficially* checking that everything works as expected can take several minutes. As we make changes and grow the site, the time required to manually check that everything works "properly" will only grow. If we were to continue as we are, eventually we'd be spending most of our time testing, and very little time improving our code.

Automated tests can really help with this problem! The obvious benefits are that they can be run much faster than manual tests, can test to a much lower level of detail, and test exactly the same functionality every time (human testers are nowhere near as reliable!) Because they are fast, automated tests can be executed more regularly, and if a test fails, they point to exactly where code is not performing as expected.

In addition, automated tests can act as the first real-world "user" of your code, forcing you to be rigorous about defining and documenting how your website should behave. Often they are basis for your code examples and documentation. For these reasons, some software development processes start with test definition and implementation, after which the code is written to match the required behavior (e.g. [test-driven](#) and [behaviour-driven](#) development).

This tutorial shows how to write automated tests for Django, by adding a number of tests to the *LocalLibrary* website.

[Types of testing](#)

There are numerous types, levels, and classifications of tests and testing approaches. The most important automated tests are:

Unit tests

Verify functional behavior of individual components, often to class and function level.

Regression tests

Tests that reproduce historic bugs. Each test is initially run to verify that the bug has been fixed, and then re-run to ensure that it has not been reintroduced following later changes to the code.

Integration tests

Verify how groupings of components work when used together. Integration tests are aware of the required interactions between components, but not necessarily of the internal operations of each component. They may cover simple groupings of components through to the whole website.

Note: Other common types of tests include black box, white box, manual, automated, canary, smoke, conformance, acceptance, functional, system, performance, load, and stress tests. Look them up for more information.

What does Django provide for testing?

Testing a website is a complex task, because it is made of several layers of logic – from HTTP-level request handling, queries models, to form validation and processing, and template rendering.

Django provides a test framework with a small hierarchy of classes that build on the Python standard [unittest](#) library. Despite the name, this test framework is suitable for both unit and integration tests. The Django framework adds API methods and tools to help test web and Django-specific behaviour. These allow you to simulate requests, insert test data, and inspect your application's output. Django also provides an API ([LiveServerTestCase](#)) and tools for [using different testing frameworks](#), for example you can integrate with the popular [Selenium](#) framework to simulate a user interacting with a live browser.

To write a test you derive from any of the Django (or *unittest*) test base classes ([SimpleTestCase](#), [TransactionTestCase](#), [TestCase](#), [LiveServerTestCase](#)) and then write separate methods to check that specific functionality works as expected (tests use "assert" methods to test that expressions result in `True` or `False` values, or that two values are equal, etc.) When you start a test run, the framework executes the chosen test methods in your derived classes. The test methods are run independently, with common setup and/or tear-down behaviour defined in the class, as shown below.

```
class YourTestClass(TestCase):

    def setUp(self):
        #Setup run before every test method.
        pass

    def tearDown(self):
        #Clean up run after every test method.
        pass

    def test_something_that_will_pass(self):
        self.assertFalse(False)
```

```
def test_something_that_will_fail(self):
    self.assertTrue(False)
```

The best base class for most tests is [django.test.TestCase](#). This test class creates a clean database before its tests are run, and runs every test function in its own transaction. The class also owns a test [Client](#) that you can use to simulate a user interacting with the code at the view level. In the following sections we're going to concentrate on unit tests, created using this [TestCase](#) base class.

Note: The [django.test.TestCase](#) class is very convenient, but may result in some tests being slower than they need to be (not every test will need to set up its own database or simulate the view interaction). Once you're familiar with what you can do with this class, you may want to replace some of your tests with the available simpler test classes.

What should you test?

You should test all aspects of your own code, but not any libraries or functionality provided as part of Python or Django.

So for example, consider the `Author` model defined below. You don't need to explicitly test that `first_name` and `last_name` have been stored properly as `CharField` in the database because that is something defined by Django (though of course in practice you will inevitably test this functionality during development). Nor do you need to test that the `date_of_birth` has been validated to be a date field, because that is again something implemented in Django.

However you should check the text used for the labels (*First name*, *Last name*, *Date of birth*, *Died*), and the size of the field allocated for the text (*100 chars*), because these are part of your design and something that could be broken/changed in future.

```
class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True, blank=True)
    date_of_death = models.DateField('Died', null=True, blank=True)

    def get_absolute_url(self):
        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        return '%s, %s' % (self.last_name, self.first_name)
```

Similarly, you should check that the custom methods `get_absolute_url()` and `__str__()` behave as required because they are your code/business logic. In the case of `get_absolute_url()` you can trust that the Django `reverse()` method has been implemented properly, so what you're testing is that the associated view has actually been defined.

Note: Astute readers may note that we would also want to constrain the date of birth and death to sensible values, and check that death comes after birth. In Django this constraint would be added to your form classes (although you can define validators for the fields these appear to only be used at the form level, not the model level).

With that in mind lets start looking at how to define and run tests.

Before we go into the detail of "what to test", let's first briefly look at *where* and *how* tests are defined.

Django uses the unittest module's [built-in test discovery](#), which will discover tests under the current working directory in any file named with the pattern **test*.py**. Provided you name the files appropriately, you can use any structure you like. We recommend that you create a module for your test code, and have separate files for models, views, forms, and any other types of code you need to test. For example:

```
catalog/  
  /tests/  
    __init__.py  
    test_models.py  
    test_forms.py  
    test_views.py
```

Create a file structure as shown above in your *LocalLibrary* project. The **__init__.py** should be an empty file (this tells Python that the directory is a package). You can create the three test files by copying and renaming the skeleton test file **/catalog/tests.py**.

Note: The skeleton test file **/catalog/tests.py** was created automatically when we [built the Django skeleton website](#). It is perfectly "legal" to put all your tests inside it, but if you test properly, you'll quickly end up with a very large and unmanageable test file.

Delete the skeleton file as we won't need it.

Open **/catalog/tests/test_models.py**. The file already imports `django.test.TestCase`, as shown:

```
from django.test import TestCase  
  
# Create your tests here.
```

Often you will add a test class for each model/view/form you want to test, with individual methods for testing specific functionality. In other cases you may wish to have a separate class for testing a specific use case, with individual test functions that test aspects of that use-case (for example, a class to test that a model field is properly validated, with functions to test each of the possible failure cases). Again, the structure is very much up to you, but it is best if you are consistent.

Add the test class below to the bottom of the file. The class demonstrates how to construct a test case class by deriving from `TestCase`.

```
class YourTestClass(TestCase):  
  
    @classmethod  
    def setUpTestData(cls):  
        print("setUpTestData: Run once to set up non-modified data for all  
class methods.")  
        pass
```



```

def setUp(self):
    print("setUp: Run once for every test method to setup clean data.")
    pass

def test_false_is_false(self):
    print("Method: test_false_is_false.")
    self.assertFalse(False)

def test_false_is_true(self):
    print("Method: test_false_is_true.")
    self.assertTrue(False)

def test_one_plus_one_equals_two(self):
    print("Method: test_one_plus_one_equals_two.")
    self.assertEqual(1 + 1, 2)

```

The new class defines two methods that you can use for pre-test configuration (for example, to create any models or other objects you will need for the test):

- `setUpTestData()` is called once at the beginning of the test run for class-level setup. You'd use this to create objects that aren't going to be modified or changed in any of the test methods.
- `setUp()` is called before every test function to set up any objects that may be modified by the test (every test function will get a "fresh" version of these objects).

The test classes also have a `tearDown()` method which we haven't used. This method isn't particularly useful for database tests, since the `TestCase` base class takes care of database teardown for you.

Below those we have a number of test methods, which use `Assert` functions to test whether conditions are true, false or equal (`AssertTrue`, `AssertFalse`, `AssertEqual`). If the condition does not evaluate as expected then the test will fail and report the error to your console.

The `AssertTrue`, `AssertFalse`, `AssertEqual` are standard assertions provided by **unittest**. There are other standard assertions in the framework, and also [Django-specific assertions](#) to test if a view redirects (`assertRedirects`), to test if a particular template has been used (`assertTemplateUsed`), etc.

You should **not** normally include `print()` functions in your tests as shown above. We do that here only so that you can see the order that the setup functions are called in the console (in the following section).

[How to run the tests](#)[Edit](#)

The easiest way to run all the tests is to use the command:

```
python3 manage.py test
```

This will discover all files named with the pattern **test*.py** under the current directory and run all tests defined using appropriate base classes (here we have a number of test files, but only **/catalog/tests/test_models.py** currently contains any tests.) By default the tests will individually report only on test failures, followed by a test summary.

Run the tests in the root directory of *LocalLibrary*. You should see an output like the one below.

```
>python manage.py test

Creating test database for alias 'default'...
setUpTestData: Run once to set up non-modified data for all class methods.
setUp: Run once for every test method to setup clean data.
Method: test_false_is_false.
.setUp: Run once for every test method to setup clean data.
Method: test_false_is_true.
FsetUp: Run once for every test method to setup clean data.
Method: test_one_plus_one_equals_two.
.
=====
FAIL: test_false_is_true (catalog.tests.tests_models.YourTestClass)
-----
Traceback (most recent call last):
  File
"D:\Github\django_tmp\library_w_t_2\locallibrary\catalog\tests\tests_models.p
y", line 22, in test_false_is_true
    self.assertTrue(False)
AssertionError: False is not true

-----
Ran 3 tests in 0.075s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Here we see that we had one test failure, and we can see exactly what function failed and why (this failure is expected, because `False` is not `True`!).

Tip: The most important thing to learn from the test output above is that it is much more valuable if you use descriptive/informative names for your objects and methods.

The text shown in **bold** above would not normally appear in the test output (this is generated by the `print()` functions in our tests). This shows how the `setUpTestData()` method is called once for the class and `setUp()` is called before each method.

The next sections show how you can run specific tests, and how to control how much information the tests display.

Showing more test information

If you want to get more information about the test run you can change the *verbosity*. For example, to list the test successes as well as failures (and a whole bunch of information about how the testing database is set up) you can set the verbosity to "2" as shown:

```
python3 manage.py test --verbosity 2
```

The allowed verbosity levels are 0, 1, 2, and 3, with the default being "1".

Running specific tests

If you want to run a subset of your tests you can do so by specifying the full dot path to the package(s), module, `TestCase` subclass or method:

```
python3 manage.py test catalog.tests      # Run the specified module
python3 manage.py test catalog.tests.test_models  # Run the specified module
python3 manage.py test catalog.tests.test_models.YourTestClass # Run the
specified class
python3 manage.py test
catalog.tests.test_models.YourTestClass.test_one_plus_one_equals_two  # Run
the specified method
```

[LocalLibrary tests](#)[Edit](#)

Now we know how to run our tests and what sort of things we need to test, let's look at some practical examples.

Note: We won't write every possible test, but this should give you an idea of how tests work, and what more you can do.

Models

As discussed above, we should test anything that is part of our design or that is defined by code that we have written, but not libraries/code that is already tested by Django or the Python development team.

For example, consider the `Author` model below. Here we should test the labels for all the fields, because even though we haven't explicitly specified most of them, we have a design that says what these values should be. If we don't test the values, then we don't know that the field labels have their intended values. Similarly while we trust that Django will create a field of the specified length, it is worthwhile to specify a test for this length to ensure that it was implemented as planned.

```
class Author(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    date_of_birth = models.DateField(null=True, blank=True)
    date_of_death = models.DateField('Died', null=True, blank=True)

    def get_absolute_url(self):
        return reverse('author-detail', args=[str(self.id)])

    def __str__(self):
        return '%s, %s' % (self.last_name, self.first_name)
```

Open our `/catalog/tests/test_models.py`, and replace any existing code with the following test code for the `Author` model.

Here you'll see that we first import `TestCase` and derive our test class (`AuthorModelTest`) from it, using a descriptive name so we can easily identify any failing tests in the test output. We then call `setUpTestData()` to create an `author` object that we will use but not modify in any of the tests.

```

from django.test import TestCase

# Create your tests here.

from catalog.models import Author

class AuthorModelTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        #Set up non-modified objects used by all test methods
        Author.objects.create(first_name='Big', last_name='Bob')

    def test_first_name_label(self):
        author=Author.objects.get(id=1)
        field_label = author._meta.get_field('first_name').verbose_name
        self.assertEqual(field_label, 'first name')

    def test_date_of_death_label(self):
        author=Author.objects.get(id=1)
        field_label = author._meta.get_field('date_of_death').verbose_name
        self.assertEqual(field_label, 'died')

    def test_first_name_max_length(self):
        author=Author.objects.get(id=1)
        max_length = author._meta.get_field('first_name').max_length
        self.assertEqual(max_length, 100)

    def test_object_name_is_last_name_comma_first_name(self):
        author=Author.objects.get(id=1)
        expected_object_name = '%s, %s' % (author.last_name,
author.first_name)
        self.assertEqual(expected_object_name, str(author))

    def test_get_absolute_url(self):
        author=Author.objects.get(id=1)
        #This will also fail if the urlconf is not defined.
        self.assertEqual(author.get_absolute_url(), '/catalog/author/1')

```

The field tests check that the values of the field labels (`verbose_name`) and that the size of the character fields are as expected. These methods all have descriptive names, and follow the same pattern:

```

author=Author.objects.get(id=1)    # Get an author object to test
field_label = author._meta.get_field('first_name').verbose_name    # Get the
metadata for the required field and use it to query the required field data
self.assertEqual(field_label, 'first name')    # Compare the value to the
expected result

```

The interesting things to note are:

- We can't get the `verbose_name` directly using `author.first_name.verbose_name`, because `author.first_name` is a *string* (not a handle to the `first_name` object that we can use to access its properties). Instead we need to use the author's `_meta` attribute to get an instance of the field and use that to query for the additional information.
- We chose to use `assertEquals(field_label, 'first name')` rather than `assertTrue(field_label == 'first name')`. The reason for this is that if the test fails

the output for the former tells you what the label actually was, which makes debugging the problem just a little easier.

Note: Tests for the `last_name` and `date_of_birth` labels, and also the test for the length of the `last_name` field have been omitted. Add your own versions now, following the naming conventions and approaches shown above.

We also need to test our custom methods. These essentially just check that the object name was constructed as we expected using "Last Name", "First Name" format, and that the URL we get for an `Author` item is as we would expect.

```
def test_object_name_is_last_name_comma_first_name(self):
    author=Author.objects.get(id=1)
    expected_object_name = '%s, %s' % (author.last_name, author.first_name)
    self.assertEqual(expected_object_name, str(author))

def test_get_absolute_url(self):
    author=Author.objects.get(id=1)
    #This will also fail if the urlconf is not defined.
    self.assertEqual(author.get_absolute_url(), '/catalog/author/1')
```

Run the tests now. If you created the `Author` model as we described in the models tutorial it is quite likely that you will get an error for the `date_of_birth` label as shown below. The test is failing because it was written expecting the label definition to follow Django's convention of not capitalising the first letter of the label (Django does this for you).

```
=====
FAIL: test_date_of_death_label (catalog.tests.test_models.AuthorModelTest)
-----
Traceback (most recent call last):
  File "D:\...\locallibrary\catalog\tests\test_models.py", line 32, in
test_date_of_death_label
    self.assertEqual(field_label, 'died')
AssertionError: 'Died' != 'died'
- Died
? ^
+ died
? ^
```

This is a very minor bug, but it does highlight how writing tests can more thoroughly check any assumptions you may have made.

Note: Change the label for the `date_of_death` field (`/catalog/models.py`) to "died" and re-run the tests.

The patterns for testing the other models are similar so we won't continue to discuss these further. Feel free to create your own tests for the our other models.

Forms

The philosophy for testing your forms is the same as for testing your models; you need to test anything that you've coded or your design specifies, but not the behaviour of the underlying framework and other third party libraries.

Generally this means that you should test that the forms have the fields that you want, and that these are displayed with appropriate labels and help text. You don't need to verify that Django validates the field type correctly (unless you created your own custom field and validation) — i.e. you don't need to test that an email field only accepts emails. However you would need to test any additional validation that you expect to be performed on the fields and any messages that your code will generate for errors.

Consider our form for renewing books. This has just one field for the renewal date, which will have a label and help text that we will need to verify.

```
class RenewBookForm(forms.Form):
    """
    Form for a librarian to renew books.
    """
    renewal_date = forms.DateField(help_text="Enter a date between now and 4
weeks (default 3).")

    def clean_renewal_date(self):
        data = self.cleaned_data['renewal_date']

        #Check date is not in past.
        if data < datetime.date.today():
            raise ValidationError(_('Invalid date - renewal in past'))
        #Check date is in range librarian allowed to change (+4 weeks)
        if data > datetime.date.today() + datetime.timedelta(weeks=4):
            raise ValidationError(_('Invalid date - renewal more than 4 weeks
ahead'))

        # Remember to always return the cleaned data.
        return data
```

Open our `/catalog/tests/test_forms.py` file and replace any existing code with the following test code for the `RenewBookForm` form. We start by importing our form and some Python and Django libraries to help test time-related functionality. We then declare our form test class in the same way as we did for models, using a descriptive name for our `TestCase`-derived test class.

```
from django.test import TestCase

# Create your tests here.

import datetime
from django.utils import timezone
from catalog.forms import RenewBookForm

class RenewBookFormTest(TestCase):

    def test_renew_form_date_field_label(self):
        form = RenewBookForm()
        self.assertTrue(form.fields['renewal_date'].label == None or
form.fields['renewal_date'].label == 'renewal date')

    def test_renew_form_date_field_help_text(self):
        form = RenewBookForm()
        self.assertEqual(form.fields['renewal_date'].help_text, 'Enter a date
between now and 4 weeks (default 3).')

    def test_renew_form_date_in_past(self):
```

```

        date = datetime.date.today() - datetime.timedelta(days=1)
        form_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        self.assertFalse(form.is_valid())

    def test_renew_form_date_too_far_in_future(self):
        date = datetime.date.today() + datetime.timedelta(weeks=4) +
datetime.timedelta(days=1)
        form_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        self.assertFalse(form.is_valid())

    def test_renew_form_date_today(self):
        date = datetime.date.today()
        form_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        self.assertTrue(form.is_valid())

    def test_renew_form_date_max(self):
        date = timezone.now() + datetime.timedelta(weeks=4)
        form_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        self.assertTrue(form.is_valid())

```

The first two functions test that the field's `label` and `help_text` are as expected. We have to access the field using the fields dictionary (e.g. `form.fields['renewal_date']`). Note here that we also have to test whether the label value is `None`, because even though Django will render the correct label it returns `None` if the value is not *explicitly* set.

The rest of the functions test that the form is valid for renewal dates just inside the acceptable range and invalid for values outside the range. Note how we construct test date values around our current date (`datetime.date.today()`) using `datetime.timedelta()` (in this case specifying a number of days or weeks). We then just create the form, passing in our data, and test if it is valid.

Note: Here we don't actually use the database or test client. Consider modifying these tests to use [SimpleTestCase](#).

We also need to validate that the correct errors are raised if the form is invalid, however this is usually done as part of view processing, so we'll take care of that in the next section.

That's all for forms; we do have some others, but they are automatically created by our generic class-based editing views, and should be tested there! Run the tests and confirm that our code still passes!

Views

To validate our view behaviour we use the Django test [Client](#). This class acts like a dummy web browser that we can use to simulate `GET` and `POST` requests on a URL and observe the response. We can see almost everything about the response, from low-level HTTP (result headers and status codes) through to the template we're using to render the HTML and the context data we're passing to it. We can also see the chain of redirects (if any) and check the URL and status code at each step. This allows us to verify that each view is doing what is expected.

Let's start with one of our simplest views, which provides a list of all Authors. This is displayed at URL **/catalog/authors/** (an URL named 'authors' in the URL configuration).

```
class AuthorListView(generic.ListView):
    model = Author
    paginate_by = 10
```

As this is a generic list view almost everything is done for us by Django. Arguably if you trust Django then the only thing you need to test is that the view is accessible at the correct URL and can be accessed using its name. However if you're using a test-driven development process you'll start by writing tests that confirm that the view displays all Authors, paginating them in lots of 10.

Open the **/catalog/tests/test_views.py** file and replace any existing text with the following test code for `AuthorListView`. As before we import our model and some useful classes. In the `setUpTestData()` method we set up a number of `Author` objects so that we can test our pagination.

```
from django.test import TestCase

# Create your tests here.

from catalog.models import Author
from django.core.urlresolvers import reverse

class AuthorListViewTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        #Create 13 authors for pagination tests
        number_of_authors = 13
        for author_num in range(number_of_authors):
            Author.objects.create(first_name='Christian %s' % author_num,
last_name = 'Surname %s' % author_num,)

    def test_view_url_exists_at_desired_location(self):
        resp = self.client.get('/catalog/authors/')
        self.assertEqual(resp.status_code, 200)

    def test_view_url_accessible_by_name(self):
        resp = self.client.get(reverse('authors'))
        self.assertEqual(resp.status_code, 200)

    def test_view_uses_correct_template(self):
        resp = self.client.get(reverse('authors'))
        self.assertEqual(resp.status_code, 200)

        self.assertTemplateUsed(resp, 'catalog/author_list.html')

    def test_pagination_is_ten(self):
        resp = self.client.get(reverse('authors'))
        self.assertEqual(resp.status_code, 200)
        self.assertTrue('is_paginated' in resp.context)
        self.assertTrue(resp.context['is_paginated'] == True)
        self.assertTrue(len(resp.context['author_list']) == 10)

    def test_lists_all_authors(self):
        #Get second page and confirm it has (exactly) remaining 3 items
```



```

resp = self.client.get(reverse('authors')+'?page=2')
self.assertEqual(resp.status_code, 200)
self.assertTrue('is_paginated' in resp.context)
self.assertTrue(resp.context['is_paginated'] == True)
self.assertTrue(len(resp.context['author_list']) == 3)

```

All the tests use the client (belonging to our `TestCase`'s derived class) to simulate a `GET` request and get a response (`resp`). The first version checks a specific URL (note, just the specific path without the domain) while the second generates the URL from its name in the URL configuration.

```

resp = self.client.get('/catalog/authors/')
resp = self.client.get(reverse('authors'))

```

Once we have the response we query it for its status code, the template used, whether or not the response is paginated, the number of items returned, and the total number of items.

The most interesting variable we demonstrate above is `resp.context`, which is the context variable passed to the template by the view. This is incredibly useful for testing, because it allows us to confirm that our template is getting all the data it needs. In other words we can check that we're using the intended template and what data the template is getting, which goes a long way to verifying that any rendering issues are solely due to template.

Views that are restricted to logged in users

In some cases you'll want to test a view that is restricted to just logged in users. For example our `LoanedBooksByUserListView` is very similar to our previous view but is only available to logged in users, and only displays `BookInstance` records that are borrowed by the current user, have the 'on loan' status, and are ordered "oldest first".

```

from django.contrib.auth.mixins import LoginRequiredMixin

class LoanedBooksByUserListView(LoginRequiredMixin, generic.ListView):
    """
    Generic class-based view listing books on loan to current user.
    """
    model = BookInstance
    template_name = 'catalog/bookinstance_list_borrowed_user.html'
    paginate_by = 10

    def get_queryset(self):
        return
        BookInstance.objects.filter(borrower=self.request.user).filter(status__exact=
        'o').order_by('due_back')

```

Add the following test code to `/catalog/tests/test_views.py`. Here we first use `SetUp()` to create some user login accounts and `BookInstance` objects (along with their associated books and other records) that we'll use later in the tests. Half of the books are borrowed by each test user, but we've initially set the status of all books to "maintenance". We've used `SetUp()` rather than `setUpTestData()` because we'll be modifying some of these objects later.

Note: The `setUp()` code below creates a book with a specified `Language`, but *your* code may not include the `Language` model as this was created as a *challenge*. If this is the case,

simply comment out the parts of the code that create or import Language objects. You should also do this in the `RenewBookInstancesViewTest` section that follows.

```
import datetime
from django.utils import timezone

from catalog.models import BookInstance, Book, Genre, Language
from django.contrib.auth.models import User #Required to assign User as a
borrower

class LoanedBookInstancesByUserListViewTest(TestCase):

    def setUp(self):
        #Create two users
        test_user1 = User.objects.create_user(username='testuser1',
password='12345')
        test_user1.save()
        test_user2 = User.objects.create_user(username='testuser2',
password='12345')
        test_user2.save()

        #Create a book
        test_author = Author.objects.create(first_name='John',
last_name='Smith')
        test_genre = Genre.objects.create(name='Fantasy')
        test_language = Language.objects.create(name='English')
        test_book = Book.objects.create(title='Book Title', summary = 'My
book summary', isbn='ABCDEFGF', author=test_author, language=test_language,)
        # Create genre as a post-step
        genre_objects_for_book = Genre.objects.all()
        test_book.genre=genre_objects_for_book
        test_book.save()

        #Create 30 BookInstance objects
        number_of_book_copies = 30
        for book_copy in range(number_of_book_copies):
            return_date= timezone.now() +
datetime.timedelta(days=book_copy%5)
            if book_copy % 2:
                the_borrower=test_user1
            else:
                the_borrower=test_user2
            status='m'
            BookInstance.objects.create(book=test_book,imprint='Unlikely
Imprint, 2016', due_back=return_date, borrower=the_borrower, status=status)

    def test_redirect_if_not_logged_in(self):
        resp = self.client.get(reverse('my-borrowed'))
        self.assertRedirects(resp, '/accounts/login/?next=/catalog/mybooks/')

    def test_logged_in_uses_correct_template(self):
        login = self.client.login(username='testuser1', password='12345')
        resp = self.client.get(reverse('my-borrowed'))

        #Check our user is logged in
        self.assertEqual(str(resp.context['user']), 'testuser1')
        #Check that we got a response "success"
        self.assertEqual(resp.status_code, 200)

        #Check we used correct template
```

```
        self.assertTemplateUsed(resp,  
'catalog/bookinstance_list_borrowed_user.html')
```

To verify that the view will redirect to a login page if the user is not logged in we use `assertRedirects`, as demonstrated in `test_redirect_if_not_logged_in()`. To verify that the page is displayed for a logged in user we first log in our test user, and then access the page again and check that we get a `status_code` of 200 (success).

The rest of the test verify that our view only returns books that are on loan to our current borrower. Copy the (self-explanatory) code at the end of the test class above.

```
def test_only_borrowed_books_in_list(self):  
    login = self.client.login(username='testuser1', password='12345')  
    resp = self.client.get(reverse('my-borrowed'))  
  
    #Check our user is logged in  
    self.assertEqual(str(resp.context['user']), 'testuser1')  
    #Check that we got a response "success"  
    self.assertEqual(resp.status_code, 200)  
  
    #Check that initially we don't have any books in list (none on loan)  
    self.assertTrue('bookinstance_list' in resp.context)  
    self.assertEqual(len(resp.context['bookinstance_list']), 0)  
  
    #Now change all books to be on loan  
    get_ten_books = BookInstance.objects.all()[:10]  
  
    for copy in get_ten_books:  
        copy.status='o'  
        copy.save()  
  
    #Check that now we have borrowed books in the list  
    resp = self.client.get(reverse('my-borrowed'))  
    #Check our user is logged in  
    self.assertEqual(str(resp.context['user']), 'testuser1')  
    #Check that we got a response "success"  
    self.assertEqual(resp.status_code, 200)  
  
    self.assertTrue('bookinstance_list' in resp.context)  
  
    #Confirm all books belong to testuser1 and are on loan  
    for bookitem in resp.context['bookinstance_list']:  
        self.assertEqual(resp.context['user'], bookitem.borrower)  
        self.assertEqual('o', bookitem.status)  
  
def test_pages_ordered_by_due_date(self):  
  
    #Change all books to be on loan  
    for copy in BookInstance.objects.all():  
        copy.status='o'  
        copy.save()  
  
    login = self.client.login(username='testuser1', password='12345')  
    resp = self.client.get(reverse('my-borrowed'))  
  
    #Check our user is logged in  
    self.assertEqual(str(resp.context['user']), 'testuser1')  
    #Check that we got a response "success"  
    self.assertEqual(resp.status_code, 200)
```

```

#Confirm that of the items, only 10 are displayed due to pagination.
self.assertEqual( len(resp.context['bookinstance_list']),10)

last_date=0
for copy in resp.context['bookinstance_list']:
    if last_date==0:
        last_date=copy.due_back
    else:
        self.assertTrue(last_date <= copy.due_back)

```

You could also add pagination tests, should you so wish!

Testing views with forms

Testing views with forms is a little more complicated than in the cases above, because you need to test more code paths: initial display, display after data validation has failed, and display after validation has succeeded. The good news is that we use the client for testing in almost exactly the same way as we did for display-only views.

To demonstrate, let's write some tests for the view used to renew books

(`renew_book_librarian()`):

```

from .forms import RenewBookForm

@permission_required('catalog.can_mark_returned')
def renew_book_librarian(request, pk):
    """
    View function for renewing a specific BookInstance by librarian
    """
    book_inst=get_object_or_404(BookInstance, pk = pk)

    # If this is a POST request then process the Form data
    if request.method == 'POST':

        # Create a form instance and populate it with data from the request
        (binding):
        form = RenewBookForm(request.POST)

        # Check if the form is valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required (here we just
            write it to the model due_back field)
            book_inst.due_back = form.cleaned_data['renewal_date']
            book_inst.save()

            # redirect to a new URL:
            return HttpResponseRedirect(reverse('all-borrowed') )

        # If this is a GET (or any other method) create the default form
        else:
            proposed_renewal_date = datetime.date.today() +
            datetime.timedelta(weeks=3)
            form = RenewBookForm(initial={'renewal_date':
            proposed_renewal_date,})

            return render(request, 'catalog/book_renew_librarian.html', {'form':
            form, 'bookinst':book_inst})

```

We'll need to test that the view is only available to users who have the `can_mark_returned` permission, and that users are redirected to an HTTP 404 error page if they attempt to renew a `BookInstance` that does not exist. We should check that the initial value of the form is seeded with a date three weeks in the future, and that if validation succeeds we're redirected to the "all-borrowed books" view. As part checking the validation-fail tests we'll also check that our form is sending the appropriate error messages.

Add the first part of the test class (shown below) to the bottom of `/catalog/tests/test_views.py`. This creates two users and two book instances, but only gives one user the permission required to access the view. The code to grant permissions during tests is shown in bold:

```
from django.contrib.auth.models import Permission # Required to grant the
permission needed to set a book as returned.

class RenewBookInstancesViewTest(TestCase):

    def setUp(self):
        #Create a user
        test_user1 = User.objects.create_user(username='testuser1',
password='12345')
        test_user1.save()

        test_user2 = User.objects.create_user(username='testuser2',
password='12345')
        test_user2.save()
permission = Permission.objects.get(name='Set book as returned')
test_user2.user_permissions.add(permission)
test_user2.save()

        #Create a book
        test_author = Author.objects.create(first_name='John',
last_name='Smith')
        test_genre = Genre.objects.create(name='Fantasy')
        test_language = Language.objects.create(name='English')
        test_book = Book.objects.create(title='Book Title', summary = 'My
book summary', isbn='ABCDEFGH', author=test_author, language=test_language,)
        # Create genre as a post-step
        genre_objects_for_book = Genre.objects.all()
        test_book.genre=genre_objects_for_book
        test_book.save()

        #Create a BookInstance object for test_user1
        return_date= datetime.date.today() + datetime.timedelta(days=5)

self.test_bookinstance1=BookInstance.objects.create(book=test_book,imprint='U
nlikely Imprint, 2016', due_back=return_date, borrower=test_user1,
status='o')

        #Create a BookInstance object for test_user2
        return_date= datetime.date.today() + datetime.timedelta(days=5)

self.test_bookinstance2=BookInstance.objects.create(book=test_book,imprint='U
nlikely Imprint, 2016', due_back=return_date, borrower=test_user2,
status='o')
```

Add the following tests to the bottom of the test class. These check that only users with the correct permissions (*testuser2*) can access the view. We check all the cases: when the user is not logged in, when a user is logged in but does not have the correct permissions, when the user has

permissions but is not the borrower (should succeed), and what happens when they try to access a BookInstance that doesn't exist. We also check that the correct template is used.

```
def test_redirect_if_not_logged_in(self):
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,})) )
    #Manually check redirect (Can't use assertRedirect, because the
redirect URL is unpredictable)
    self.assertEqual( resp.status_code,302)
    self.assertTrue( resp.url.startswith('/accounts/login/') )

def test_redirect_if_logged_in_but_not_correct_permission(self):
    login = self.client.login(username='testuser1', password='12345')
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,})) )

    #Manually check redirect (Can't use assertRedirect, because the
redirect URL is unpredictable)
    self.assertEqual( resp.status_code,302)
    self.assertTrue( resp.url.startswith('/accounts/login/') )

def test_logged_in_with_permission_borrowed_book(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance2.pk,})) )

    #Check that it lets us login - this is our book and we have the right
permissions.
    self.assertEqual( resp.status_code,200)

def test_logged_in_with_permission_another_users_borrowed_book(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,})) )

    #Check that it lets us login. We're a librarian, so we can view any
users book
    self.assertEqual( resp.status_code,200)

def test_HTTP404_for_invalid_book_if_logged_in(self):
    import uuid
    test_uid = uuid.uuid4() #unlikely UID to match our bookinstance!
    login = self.client.login(username='testuser2', password='12345')
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':test_uid,})) )
    self.assertEqual( resp.status_code,404)

def test_uses_correct_template(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,})) )
    self.assertEqual( resp.status_code,200)

    #Check we used correct template
    self.assertTemplateUsed(resp, 'catalog/book_renew_librarian.html')
```

Add the next test method, as shown below. This checks that the initial date for the form is three weeks in the future. Note how we are able to access the value of the initial value of the form field (shown in **bold**).

```

def
test_form_renewal_date_initially_has_date_three_weeks_in_future(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = self.client.get(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,}))
    self.assertEqual( resp.status_code,200)

    date_3_weeks_in_future = datetime.date.today() +
datetime.timedelta(weeks=3)
    self.assertEqual(resp.context['form'].initial['renewal_date'],
date_3_weeks_in_future )

```

The next test (add this to the class too) checks that the view redirects to a list of all borrowed books if renewal succeeds. What differs here is that for the first time we show how you can `POST` data using the client. The post *data* is the second argument to the post function, and is specified as a dictionary of key/values.

```

def test_redirects_to_all_borrowed_book_list_on_success(self):
    login = self.client.login(username='testuser2', password='12345')
    valid_date_in_future = datetime.date.today() +
datetime.timedelta(weeks=2)
    resp = self.client.post(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,}),
{'renewal_date':valid_date_in_future} )
    self.assertRedirects(resp, reverse('all-borrowed') )

```

The *all-borrowed* view was added as a *challenge*, and your code may instead redirect to the home page `'/'`. If so, modify the last two lines of the test code to be like the code below. The `follow=True` in the request ensures that the request returns the final destination URL (hence checking `/catalog/` rather than `/`).

```

resp = self.client.post(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,}),
{'renewal_date':valid_date_in_future},follow=True )
self.assertRedirects(resp, '/catalog/')

```

Copy the last two functions into the class, as seen below. These again test `POST` requests, but in this case with invalid renewal dates. We use `assertFormError()` to verify that the error messages are as expected.

```

def test_form_invalid_renewal_date_past(self):
    login = self.client.login(username='testuser2',
password='12345')
    date_in_past = datetime.date.today() - datetime.timedelta(weeks=1)
    resp = self.client.post(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,}), {'renewal_date':date_in_past})
    self.assertEqual( resp.status_code,200)
    self.assertFormError(resp, 'form', 'renewal_date', 'Invalid date -
renewal in past')

def test_form_invalid_renewal_date_future(self):
    login = self.client.login(username='testuser2', password='12345')
    invalid_date_in_future = datetime.date.today() +
datetime.timedelta(weeks=5)
    resp = self.client.post(reverse('renew-book-librarian',
kwargs={'pk':self.test_bookinstance1.pk,}),
{'renewal_date':invalid_date_in_future} )

```

```
self.assertEqual( resp.status_code,200)
self.assertFormError(resp, 'form', 'renewal_date', 'Invalid date -
renewal more than 4 weeks ahead')
```

The same sorts of techniques can be used to test the other view.

Templates

Django provides test APIs to check that the correct template is being called by your views, and to allow you to verify that the correct information is being sent. There is however no specific API support for testing in Django that your HTML output is rendered as expected.

Other recommended test tools[Edit](#)

Django's test framework can help you write effective unit and integration tests — we've only scratched the surface of what the underlying **unittest** framework can do, let alone Django's additions (for example, check out how you can use [unittest.mock](#) to patch third party libraries so you can more thoroughly test your own code).

While there are numerous other test tools that you can use, we'll just highlight two:

- [Coverage](#): This Python tool reports on how much of your code is actually executed by your tests. It is particularly useful when you're getting started, and you are trying to work out exactly what you should test.
- [Selenium](#) is a framework to automate testing in a real browser. It allows you to simulate a real user interacting with the site, and provides a great framework for system testing your site (the next step up from integration testing).

Challenge yourself[Edit](#)

There are a lot more models and views we can test. As a simple task, try to create a test case for the `AuthorCreate` view.

```
class AuthorCreate(PermissionRequiredMixin, CreateView):
    model = Author
    fields = '__all__'
    initial={'date_of_death':'12/10/2016',}
    permission_required = 'catalog.can_mark_returned'
```

Remember that you need to check anything that you specify or that is part of the design. This will include who has access, the initial date, the template used, and where the view redirects on success.

Summary[Edit](#)

Writing test code is neither fun nor glamorous, and is consequently often left to last (or not at all) when creating a website. It is however an essential part of making sure that your code is safe to release after making changes, and cost-effective to maintain.

In this tutorial we've shown you how to write and run tests for your models, forms, and views. Most importantly we've provided a brief summary of what you should test, which is often the

hardest thing to work out when your getting started. There is a lot more to know, but even with what you've learned already you should be able to create effective unit tests for your websites.

The next and final tutorial shows how you can deploy your wonderful (and fully tested!) Django website.

Django Tutorial Part 11: Deploying Django to production

Now you've created (and tested) an awesome [LocalLibrary](#) website, you're going to want to install it on a public web server so that it can be accessed by library staff and members over the Internet. This article provides an overview of how you might go about finding a host to deploy your website, and what you need to do in order to get your site ready for production.

Prerequisites: Complete all previous tutorial topics, including [Django Tutorial Part 10: Testing a Django web application](#).

Objective: To learn where and how you can deploy a Django app to production.

Overview[Edit](#)

Once your site is finished (or finished "enough" to start public testing) you're going to need to host it somewhere more public and accessible than your personal development computer.

Up to now you've been working in a development environment, using the Django development web server to share your site to the local browser/network, and running your website with (insecure) development settings that expose debug and other private information. Before you can host a website externally you're first going to have to:

- Make a few changes to your project settings.
- Choose an environment for hosting the Django app.
- Choose an environment for hosting any static files.
- Set up a production-level infrastructure for serving your website.

This tutorial provides some guidance on your options for choosing a hosting site, a brief overview of what you need to do in order to get your Django app ready for production, and a worked example of how to install the LocalLibrary website onto the [Heroku](#) cloud hosting service.

What is a production environment?[Edit](#)

The production environment is the environment provided by the server computer where you will run your website for external consumption. The environment includes:

- Computer hardware on which the website runs.
- Operating system (e.g. Linux, Windows).
- Programming language runtime and framework libraries on top of which your website is written.
- Web server used to serve pages and other content (e.g. Nginx, Apache).
- Application server that passes "dynamic" requests between your Django website and the webserver.
- Databases on which your website is dependent.

Note: Depending on how your production is configured you might also have a reverse proxy, load balancer, etc.

The server computer could be located on your premises and connected to the Internet by a fast link, but it is far more common to use a computer that is hosted "in the cloud". What this actually means is that your code is run on some remote computer (or possibly a "virtual" computer) in your hosting company's data center(s). The remote server will usually offer some guaranteed level of computing resources (e.g. CPU, RAM, storage memory, etc.) and Internet connectivity for a certain price.

This sort of remotely accessible computing/networking hardware is referred to as *Infrastructure as a Service (IaaS)*. Many IaaS vendors provide options to preinstall a particular operating system, onto which you must install the other components of your production environment. Other vendors allow you to select more fully-featured environments, perhaps including a complete Django and web-server setup.

Note: Pre-built environments can make setting up your website very easy because they reduce the configuration, but the available options may limit you to an unfamiliar server (or other components) and may be based on an older version of the OS. Often it is better to install components yourself, so that you get the ones that you want, and when you need to upgrade parts of the system, you have some idea where to start!

Other hosting providers support Django as part of a *Platform as a Service (PaaS)* offering. In this sort of hosting you don't need to worry about most of your production environment (web server, application server, load balancers) as the host platform takes care of those for you (along with most of what you need to do in order to scale your application). That makes deployment quite easy, because you just need to concentrate on your web application and not all the other server infrastructure.

Some developers will choose the increased flexibility provided by IaaS over PaaS, while others will appreciate the reduced maintenance overhead and easier scaling of PaaS. When you're getting started, setting up your website on a PaaS system is much easier, and so that is what we'll do in this tutorial.

Tip: If you choose a Python/Django-friendly hosting provider they should provide instructions on how to set up a Django website using different configurations of webserver, application server, reverse proxy, etc (this won't be relevant if you choose a PaaS). For example, there are many step-by-step guides for various configurations in the [Digital Ocean Django community docs](#).

Choosing a hosting provider [Edit](#)

There are well over 100 hosting providers that are known to either actively support or work well with Django (you can find a fairly extensive list at [Djangofriendly hosts](#)). These vendors provide different types of environments (IaaS, PaaS), and different levels of computing and network resources at different prices.

Some of the things to consider when choosing a host:

- How busy your site is likely to be and the cost of data and computing resources required to meet that demand.
- Level of support for scaling horizontally (adding more machines) and vertically (upgrading to more powerful machines) and the costs of doing so.

- Where the supplier has data centres, and hence where access is likely to be fastest.
- The host's historical uptime and downtime performance.
- Tools provided for managing the site — are they easy to use and are they secure (e.g. SFTP vs FTP).
- Inbuilt frameworks for monitoring your server.
- Known limitations. Some hosts will deliberately block certain services (e.g. email) . Others offer only a certain number of hours of "live time" in some price tiers, or only offer a small amount of storage.
- Additional benefits. Some providers will offer free domain names and support for SSL certificates that you would otherwise have to pay for.
- Whether the "free" tier you're relying on expires over time, and whether the cost of migrating to a more expensive tier means you would have been better off using some other service in the first place!

The good news when you're starting out is that there are quite a few sites that provide "evaluation", "developer", or "hobbyist" computing environments for "free". These are always fairly resource constrained/limited environments, and you do need to be aware that they may expire after some introductory period. They are however great for testing low traffic sites in a real environment, and can provide an easy migration to paying for more resources when your site gets busier. Popular choices in this category include [Heroku](#), [Python Anywhere](#), [Amazon Web Services](#), [Microsoft Azure](#), etc.

Many providers also have a "basic" tier that provides more useful levels of computing power and fewer limitations. [Digital Ocean](#) and [Python Anywhere](#) are examples of popular hosting providers that offer a relatively inexpensive basic computing tier (in the \$5 to \$10USD per month range).

Note: Remember that price is not the only selection criteria. If your website is successful, it may turn out that scalability is the most important consideration.

Getting your website ready to publish [Edit](#)

The [Django skeleton website](#) created using the *django-admin* and *manage.py* tools are configured to make development easier. Many of the Django project settings (specified in **settings.py**) should be different for production, either for security or performance reasons.

Tip: It is common to have a separate **settings.py** file for production, and to import sensitive settings from a separate file or an environment variable. This file should then be protected, even if the rest of the source code is available on a public repository.

The critical settings that you must check are:

- `DEBUG`. This should be set as `False` in production (`DEBUG = False`). This stops the sensitive/confidential debug trace and variable information from being displayed.
- `SECRET_KEY`. This is a large random value used for CSRF protection etc. It is important that the key used in production is not in source control or accessible outside the production server. The Django documents suggest that this might best be loaded from an environment variable or read from a serve-only file.
- `# Read SECRET_KEY from an environment variable`
- `import os`

- SECRET_KEY = os.environ['SECRET_KEY']
-
- #OR
-
- #Read secret key from a file
- with open('/etc/secret_key.txt') as f:
SECRET_KEY = f.read().strip()
-

Let's change the *LocalLibrary* application so that we read our SECRET_KEY and DEBUG variables from environment variables if they are defined, but otherwise use the default values in the configuration file.

Open */locallibrary/settings.py*, disable the original SECRET_KEY configuration and add the new lines as shown below in **bold**. During development no environment variable will be specified for the key, so the default value will be used (it shouldn't matter what key you use here, or if the key "leaks", because you won't use it in production).

```
# SECURITY WARNING: keep the secret key used in production secret!
# SECRET_KEY = 'cg#p$g+j9tax!#a3cup@1$8obt2_+&k3q+pmu)5%asj6yjpgag'
import os
SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY',
'cg#p$g+j9tax!#a3cup@1$8obt2_+&k3q+pmu)5%asj6yjpgag')
```

Then comment out the existing DEBUG setting and add the new line shown below.

```
# SECURITY WARNING: don't run with debug turned on in production!
# DEBUG = True
DEBUG = bool( os.environ.get('DJANGO_DEBUG', True) )
```

The value of the DEBUG will be True by default, but will be False if the value of the DJANGO_DEBUG environment variable is set to an empty string, e.g. DJANGO_DEBUG=' '.

Note: It would be more intuitive if we could just set and unset the DJANGO_DEBUG environment variable to True and False directly, rather than using "any string" or "empty string" (respectively). Unfortunately environment variable values are stored as Python strings, and the only string that evaluates as False is the empty string (e.g. bool('')==False).

A full checklist of settings you might want to change is provided in [Deployment checklist](#) (Django docs). You can also list a number of these using the terminal command below:

```
python3 manage.py check --deploy
```

[Example: Installing LocalLibrary on Heroku](#)[Edit](#)

This section provides a practical demonstration of how to install *LocalLibrary* on the [Heroku PaaS cloud](#).

Why Heroku?

Heroku is one of the longest running and popular cloud-based PaaS services. It originally supported only Ruby apps, but now can be used to host apps from many programming environments, including Django!

We are choosing to use Heroku for several reasons:

- Heroku has a [free tier](#) that is *really* free (albeit with some limitations).
- As a PaaS, Heroku takes care of a lot of the web infrastructure for us. This makes it much easier to get started, because you don't worry about servers, load balancers, reverse proxies, or any of the other web infrastructure that Heroku provides for us under the hood.
- While it does have some limitations these will not affect this particular application. For example:
 - Heroku provides only short-lived storage so user-uploaded files cannot safely be stored on Heroku itself.
 - The free tier will sleep an inactive web app if there are no requests within a half hour period. The site may then take several seconds to respond when it is woken up.
 - The free tier limits the time that your site is running to a certain amount of hours every month (not including the time that the site is "asleep"). This is fine for a low use/demonstration site, but will not be suitable if 100% uptime is required.
 - Other limitations are listed in [Limits](#) (Heroku docs).
- Mostly it just works, and if you end up loving it, scaling your app is very easy.

While Heroku is perfect for hosting this demonstration it may not be perfect for your real website. Heroku makes things easy to set up and scale, at the cost of being less flexible, and potentially a lot more expensive once you get out of the free tier.

How does Heroku work?

Heroku runs Django websites within one or more "[Dynos](#)", which are isolated, virtualized Unix containers that provide the environment required to run an application. The dynos are completely isolated and have an *ephemeral* file system (a short-lived file system that is cleaned/emptied every time the dyno restarts). The only thing that dynos share by default are application [configuration variables](#). Heroku internally uses a load balancer to distribute web traffic to all "web" dynos. Since nothing is shared between them, Heroku can scale an app horizontally simply by adding more dynos (though of course you may also need to scale your database to accept additional connections).

Because the file system is ephemeral you can't install services required by your application directly (e.g. databases, queues, caching systems, storage, email services, etc). Instead Heroku web applications use backing services provided as independent "add-ons" by Heroku or 3rd parties. Once attached to your web application, the dynos access the services using information contained in application configuration variables.

In order to execute your application Heroku needs to be able to set up the appropriate environment and dependencies, and also understand how it is launched. For Django apps we provide this information in a number of text files:

- **runtime.txt**: the programming language and version to use.
- **requirements.txt**: the Python component dependencies, including Django.

- **Procfile:** A list of processes to be executed to start the web application. For Django this will usually be the Gunicorn web application server (with a `.wsgi` script).
- **wsgi.py:** [WSGI](#) configuration to call our Django application in the Heroku environment.

Developers interact with Heroku using a special client app/terminal, which is much like a Unix bash script. This allows you to upload code that is stored in a git repository, inspect the running processes, see logs, set configuration variables and much more!

In order to get our application to work on Heroku we'll need to put our Django web application into a git repository, add the files above, integrate with a database add-on, and make changes to properly handle static files.

Once we've done all that we can set up a Heroku account, get the Heroku client, and use it to install our website.

Note: The instructions below reflect how to work with Heroku at time of writing. If Heroku significantly change their processes, you may wish to instead check their setup documents: [Getting Started on Heroku with Django](#).

That's all the overview you need in order to get started (see [How Heroku works](#) for a more comprehensive guide).

Creating an application repository in Github

Heroku is closely integrated with the **git** source code version control system, using it to upload/synchronise any changes you make to the live system. It does this by adding a new heroku "remote" repository named *heroku* pointing to a repository for your source on the Heroku cloud. During development you use git to store changes on your "master" repository. When you want to deploy your site, you sync your changes to the Heroku repository.

Note: If you're used to following good software development practices you are probably already using git or some other SCM system. If you already have a git repository, then you can skip this step.

There are a lot of ways of to work with git, but one of the easiest is to first set up an account on [Github](#), create the repository there, and then sync to it locally:

1. Visit <https://github.com/> and create an account.
2. Once you are logged in, click the + link in the top toolbar and select **New repository**.
3. Fill in all the fields on this form. While these are not compulsory, they are strongly recommended.
 - Enter a new repository name (e.g. *django_local_library*), and description (e.g. "Local Library website written in Django").
 - Choose **Python** in the *Add .gitignore* selection list.
 - Choose your preferred license in the *Add license* selection list.
 - Check **Initialize this repository with a README**.
4. Press **Create repository**.
5. Click the green "**Clone or download**" button on your new repo page.
6. Copy the URL value from the text field inside the dialog box that appears (it should be something like: **https://github.com/<your_git_user_id>/django_local_library.git**).

Now the repository ("repo") is created we are going to want to clone it on our local computer:

1. Install *git* for your local computer (you can find versions for different platforms [here](#)).
2. Open a command prompt/terminal and clone your repository using the URL you copied above:

```
git clone
https://github.com/<your_git_user_id>/django_local_library.git
```

- This will create the repository below the current point.

- Navigate into the new repo.

```
cd django_local_library.git
```

3.

The final step is to copy in your application and then add the files to your repo using git:

1. Copy your Django application into this folder (all the files at the same level as **manage.py** and below, **not** their containing locallibrary folder).
2. Open the **.gitignore** file, copy the following lines into the bottom of it, and then save (this file is used to identify files that should not be uploaded to git by default).

```
3. # Text backup files
4. *.bak
5.
6. #Database
   *.sqlite3
```

- • Open a command prompt/terminal and use the `add` command to add all files to git.

```
git add -A
```

- • Use the `status` command to check all files that you are about to add are correct (you want to include source files, not binaries, temporary files etc.). It should look a bit like the listing below.

```
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   .gitignore
    new file:   catalog/__init__.py
    ...
    new file:   catalog/migrations/0001_initial.py
    ...
    new file:   templates/registration/password_reset_form.html
```

- • When you're satisfied commit the files to your local repository:

```
git commit -m "First version of application moved into github"
```

- • Then synchronise your local repository to the Github website, using the following:

```
git push origin master
```

6.

When this operation completes, you should be able to go back to the page on Github where you created your repo, refresh the page, and see that your whole application has now been uploaded. You can continue to update your repository as files change using this add/commit/push cycle.

Tip: This is a good point to make a backup of your "vanilla" project — while some of the changes we're going to be making in the following sections might be useful for deployment on any platform (or development) others might not.

The *best* way to do this is to use *git* to manage your revisions. With *git* you can not only go back to a particular old version, but you can maintain this in a separate "branch" from your production changes and cherry-pick any changes to move between production and development branches.

[Learning Git](#) is well worth the effort, but is beyond the scope of this topic.

The *easiest* way to do this is to just copy your files into another location. Use whichever approach best matches your knowledge of git!

Update the app for Heroku

This section explains the changes you'll need to make to our *LocalLibrary* application to get it to work on Heroku. While Heroku's [Getting Started on Heroku with Django](#) instructions assume you will use the Heroku client to also run your local development environment, our changes here are compatible with the existing Django development server and ways of working we've already learned.

Procfile

Create the file `Procfile` (no extension) in the root of your GitHub repository to declare the application's process types and entry points. Copy the following text into it:

```
web: gunicorn locallibrary.wsgi --log-file -
```

The "web:" tells Heroku that this is a web dyno and can be sent HTTP traffic. The process to start in this dyno is *gunicorn*, which is a popular web application server that Heroku recommends. We start Gunicorn using the configuration information in the module `locallibrary.wsgi` (created with our application skeleton: `/locallibrary/wsgi.py`).

Gunicorn

[Gunicorn](#) is the recommended HTTP server for use with Django on Heroku (as referenced in the Procfile above). It is a pure-Python HTTP server for WSGI applications that can run multiple Python concurrent processes within a single dyno (see [Deploying Python applications with Gunicorn](#) for more information).

While we won't need *Gunicorn* to serve our *LocalLibrary* application during development, we'll install it so that it becomes part of our [requirements](#) for Heroku to set up on the remote server.

Install *Gunicorn* locally on the command line using *pip* (which we installed when [setting up the development environment](#)):

```
pip3 install gunicorn
```

We can't use the default SQLite database on Heroku because it is file-based, and it would be deleted from the *ephemeral* file system every time the application restarts (typically once a day, and every time the application or its configuration variables are changed).

The Heroku mechanism for handling this situation is to use a [database add-on](#) and configure the web application using information from an environment [configuration variable](#), set by the add-on. There are quite a lot of database options, but we'll use the [hobby tier](#) of the *Heroku postgres* database as this is free, supported by Django, and automatically added to our new Heroku apps when using the free hobby dyno plan tier.

The database connection information is supplied to the web dyno using a configuration variable named `DATABASE_URL`. Rather than hard-coding this information into Django, Heroku recommends that developers use the [dj-database-url](#) package to parse the `DATABASE_URL` environment variable and automatically convert it to Django's desired configuration format. In addition to installing the *dj-database-url* package we'll also need to install [psycopg2](#), as Django needs this to interact with Postgres databases.

[dj-database-url](#) (Django database configuration from environment variable)

Install *dj-database-url* locally so that it becomes part of our [requirements](#) for Heroku to set up on the remote server:

```
$ pip3 install dj-database-url
settings.py
```

Open `/locallibrary/settings.py` and copy the following configuration into the bottom of the file:

```
# Heroku: Update database configuration from $DATABASE_URL.
import dj_database_url
db_from_env = dj_database_url.config(conn_max_age=500)
DATABASES['default'].update(db_from_env)
```

Note:

- We'll still be using SQLite during development because the `DATABASE_URL` environment variable will not be set on our development computer.
- The value `conn_max_age=500` makes the connection persistent, which is far more efficient than recreating the connection on every request cycle. This is however optional, and can be removed if needed.

[psycopg2](#) (Python Postgres database support)

Django needs *psycopg2* to work with Postgres databases and you will need to add this to the [requirements.txt](#) for Heroku to set this up on the remote server (as discussed in the requirements section below).

Django will use our SQLite database locally by default, because the `DATABASE_URL` environment variable isn't set in our local environment. If you want to switch to Postgres completely and use

our Heroku free tier database for both development and production then you can. For example, to install `psycopg2` and its dependencies locally on a Linux-based system you would use the following bash/terminal commands:

```
sudo apt-get install python-pip python-dev libpq-dev postgresql postgresql-contrib
pip3 install psycopg2
```

Installation instructions for the other platforms can be found on the [psycopg2 website here](#).

However, you don't need to do this — you don't need PostgreSQL active on the local computer, as long as you give it to Heroku as a requirement, in `requirements.txt` (see below).

Serving static files in production

During development we used Django and the Django development web server to serve our static files (CSS, JavaScript, etc.). In a production environment we instead typically serve static files from a content delivery network (CDN) or the web server.

Note: Serving static files via Django/web application is inefficient because the requests have to pass through unnecessary additional code (Django) rather than being handled directly by the web server or a completely separate CDN. While this doesn't matter for local use during development, it would have a significant performance impact if we were to use the same approach in production.

To make it easy to host static files separately from the Django web application, Django provides the `collectstatic` tool to collect these files for deployment (there is a settings variable that defines where the files should be collected when `collectstatic` is run). Django templates refer to the hosting location of the static files relative to a settings variable (`STATIC_URL`), so that this can be changed if the static files are moved to another host/server.

The relevant setting variables are:

- `STATIC_URL`: This is the base URL location from which static files will be served, for example on a CDN. This is used for the static template variable that is accessed in our base template (see [Django Tutorial Part 5: Creating our home page](#)).
- `STATIC_ROOT`: This is the absolute path to a directory where Django's "collectstatic" tool will gather any static files referenced in our templates. Once collected, these can then be uploaded as a group to wherever the files are to be hosted.
- `STATICFILES_DIRS`: This lists additional directories that Django's collectstatic tool should search for static files.

settings.py

Open `/locallibrary/settings.py` and copy the following configuration into the bottom of the file. The `BASE_DIR` should already have been defined in your file (the `STATIC_URL` may already have been defined within the file when it was created. While it will cause no harm, you might as well delete the duplicate previous reference).

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.10/howto/static-files/
```

```
# The absolute path to the directory where collectstatic will collect static
files for deployment.
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

# The URL to use when referring to static files (where they will be served
from)
STATIC_URL = '/static/'
```

We'll actually do the file serving using a library called [WhiteNoise](#), which we install and configure in the next section.

For more information, see [Django and Static Assets](#) (Heroku docs).

[Whitenoise](#)

There are many ways to serve static files in production (we saw the relevant Django settings in the previous sections). Heroku recommends using the [WhiteNoise](#) project for serving of static assets directly from Gunicorn in production.

Note: Heroku automatically calls *collectstatic* and prepares your static files for use by WhiteNoise after it uploads your application. Check out [WhiteNoise](#) documentation for an explanation of how it works and why the implementation is a relatively efficient method for serving these files.

The steps to set up *WhiteNoise* to use with the project are:

[WhiteNoise](#)

Install whitenoise locally using the following command:

```
$ pip3 install whitenoise
wsgi.py
```

To install *WhiteNoise* into your Django application, open `/locallibrary/wsgi.py` and copy the following text into the bottom of the file.

```
from django.core.wsgi import get_wsgi_application
from whitenoise.django import DjangoWhiteNoise

application = get_wsgi_application()
application = DjangoWhiteNoise(application)
```

This **wsgi.py** module is passed to the *Gunicorn* application server when it is started (see the [Procfile](#) above). The module provides an `application` callable that Gunicorn can use to communicate with Django, and we further configure the callable using `DjangoWhiteNoise` to serve the static files.

Note: The lines we add above duplicate some code that is already in the file. While the duplicate lines can be deleted if you wish, we don't do so in order to simplify the instructions. For more information about WSGI see [How to deploy with WSGI](#) (Django docs).

[settings.py](#)

Optionally, you can reduce the size of the static files when they are served (this is more efficient). Just add the following to the bottom of **/locallibrary/settings.py**:

```
# Simplified static file serving.  
# https://warehouse.python.org/project/whitenoise/  
STATICFILES_STORAGE = 'whitenoise.django.GzipManifestStaticFilesStorage'
```

[Requirements](#)

The Python requirements of your web application must be stored in a file **requirements.txt** in the root of your repository. Heroku will then install these automatically when it rebuilds your environment. You can create this file using *pip* on the command line (run the following in the repo root):

```
pip3 freeze > requirements.txt
```

After installing all the different dependencies above, your **requirements.txt** file should have *at least* these items listed (though the version numbers may be different). Please delete any other dependencies not listed below, unless you've explicitly added them for this application.

```
dj-database-url==0.4.1  
Django==1.10.2  
gunicorn==19.6.0  
psycopg2==2.6.2  
whitenoise==3.2.2
```

Make sure that a **psycopg2** line like the one above is present! Even if you didn't install this locally then you should still add this to the **requirements.txt**.

[Runtime](#)

The **runtime.txt** file, if defined, tells Heroku which programming language to use. Create the file in the root of the repo and add the following text:

```
python-3.5.2
```

Note: Heroku only supports a small number of [Python runtimes](#). You can specify other Python 3 runtime values, but at time of writing the above version will actually be supported for definite.

[Save changes to Github and re-test](#)

Next lets save all our changes to Github. In the terminal (whilst inside our repository), enter the following commands:

```
git add -A  
git commit -m "Added files and changes required for deployment to heroku"  
git push origin master
```

Before we proceed, lets test the site again locally and make sure it wasn't affected by any of our changes above. Run the development web server as usual and then check the site still works as you expect on your browser.

```
python3 manage.py runserver
```

We should now be ready to start deploying LocalLibrary on Heroku.

[Get a Heroku account](#)

To start using Heroku you will first need to create an account:

- Go to www.heroku.com and click the **SIGN UP FOR FREE** button.
- Enter your details and then press **CREATE FREE ACCOUNT**. You'll be asked to check your account for a sign-up email.
- Click the account activation link in the signup email. You'll be taken back to your account on the web browser.
- Enter your password and click **SET PASSWORD AND LOGIN**.
- You'll then be logged in and taken to the Heroku dashboard:
<https://dashboard.heroku.com/apps>.

[Install the client](#)

Download and install the Heroku client by following the [instructions on Heroku here](#).

After the client is installed you will be able run commands. For example to get help on the client:

```
heroku help
```

[Create and upload the website](#)

To create the app we run the "create" command in the root directory of our repository. This creates a git remote ("pointer to a remote repository") named *heroku* in our local git environment.

```
heroku create
```

Note: You can name the remote if you like by specifying a value after "create". If you don't then you'll get a random name. The name is used in the default URL.

We can then push our app to the Heroku repository as shown below. This will upload the app, package it in a dyno, run collectstatic, and start the site.

```
git push heroku master
```

If we're lucky, the app is now "running" on the site, but it won't be working properly because we haven't set up the database tables for use by our application. To do this we need to use the `heroku run` command and start a "[one off dyno](#)" to perform a migrate operation. Enter the following command in your terminal:

```
heroku run python manage.py migrate
```

We're also going to need to be able to add books and authors, so lets also create our administration superuser, again using a one-off dyno:

```
heroku run python manage.py createsuperuser
```

Once this is complete, we can look at the site. It should work, although it won't have any books in it yet. To open your browser to the new website, use the command:

```
heroku open
```

Create some books in the admin site, and check out whether the site is behaving as you expect.

Managing addons

You can check out the add-ons to your app using the `heroku addons` command. This will list all addons, and their price tier and state.

```
>heroku addons
```

Add-on	Plan	Price	State
heroku-postgresql (postgresql-flat-26536) └─ as DATABASE	hobby-dev	free	created

Here we see that we have just one add-on, the postgres SQL database. This is free, and was created automatically when we created the app. You can open a web page to examine the database add-on (or any other add-on) in more detail using the following command:

```
heroku addons:open heroku-postgresql
```

Other commands allow you to create, destroy, upgrade and downgrade addons (using a similar syntax to opening). For more information see [Managing Add-ons](#) (Heroku docs).

Setting configuration variables

You can check out the configuration variables for the site using the `heroku config` command. Below you can see that we have just one variable, the `DATABASE_URL` used to configure our database.

```
>heroku config
```

```
=== locallibrary Config Vars
DATABASE_URL: postgres://uzfnbcyxidzgrl:j2jkUFDF6OGGqxkkg7Hk3ilbZI@ec2-54-243-201-144.compute-1.amazonaws.com:5432/dbftm4qgh3kda3
```

If you recall from the section on [getting the website ready to publish](#), we have to set environment variables for `DJANGO_SECRET_KEY` and `DJANGO_DEBUG`. Let's do this now.

Note: The secret key needs to be really secret! One way to generate a new key is to create a new Django project (`django-admin startproject someprojectname`) and then get the key that is generated for you from its **settings.py**.

We set `DJANGO_SECRET_KEY` using the `config:set` command (as shown below). Remember to use your own secret key!

```
>heroku config:set DJANGO_SECRET_KEY=eu09(ilk6@4sfdofb=b_2ht@vad*$ehh9-
)3u_83+y%(+phh&=
```

```
Setting DJANGO_SECRET_KEY and restarting locallibrary... done, v7
DJANGO_SECRET_KEY: eu09(ilk6@4sfdofb=b_2ht@vad*$ehh9-)3u_83+y%(+phh
```

We similarly set DJANGO_DEBUG:

```
>heroku config:set DJANGO_DEBUG=''
```

```
Setting DJANGO_DEBUG and restarting locallibrary... done, v8
```

If you visit the site now you'll get a "Bad request" error, because the [ALLOWED_HOSTS](#) setting is *required* if you have DEBUG=False (as a security measure). Open **/locallibrary/settings.py** and change the ALLOWED_HOSTS setting to include your base app url (e.g. 'locallibrary1234.herokuapp.com') and the URL you normally use on your local development server.

```
ALLOWED_HOSTS = ['<your app URL without the https://
prefix>.herokuapp.com', '127.0.0.1']
# For example:
# ALLOWED_HOSTS = ['fathomless-scrubland-30645.herokuapp.com', '127.0.0.1']
```

Then save your settings and commit them to your Github repo and to Heroku:

```
git add -A
git commit -m 'Update ALLOWED_HOSTS with site and development server URL'
git push origin master
git push heroku master
```

After the site update to Heroku completes, enter an URL that does not exist (e.g. **/catalog/doesnotexist/**). Previously this would have displayed a detailed debug page, but now you should just see a simple "Not Found" page.

Debugging

The Heroku client provides a few tools for debugging:

```
heroku logs # Show current logs
heroku logs --tail # Show current logs and keep updating with any new results
heroku config:set DEBUG_COLLECTSTATIC=1 # Add additional logging for
collectstatic (this tool is run automatically during a build)
heroku ps #Display dyno status
```

If you need more information than these can provide you will need to start looking into [Django Logging](#).

Summary[Edit](#)

That's the end of this tutorial on setting up Django apps in production, and also the series of tutorials on working with Django. We hope you've found them useful. You can check out a fully worked-through version of the [source code on Github here](#).

The next step is to read our last few articles, and then complete the assessment task.

Page url: <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Deployment>

Django web application security

Protecting user data is an essential part of any website design. We previously explained some of the more common security threats in the article [Web security](#) — this article provides a practical demonstration of how Django's in-built protections handle such threats.

Prerequisites: Read the Server-side programming "[Website security](#)" topic. Complete the Django tutorial topics up to (and including) at least [Django Tutorial Part 9: Working with forms](#).

Objective: To understand the main things you need to do (or not do) to secure your Django web application.

[Overview](#)[Edit](#)

The [Website security](#) topic provides an overview of what website security means for server-side design, and some of the more common threats that you may need to protect against. One of the key messages in that article is that almost all attacks are successful when the web application trusts data from the browser.

Important: The single most important lesson you can learn about website security is to **never trust data from the browser**. This includes `GET` request data in URL parameters, `POST` data, HTTP headers and cookies, user-uploaded files, etc. Always check and sanitize all incoming data. Always assume the worst.

The good news for Django users is that many of the more common threats are handled by the framework! The [Security in Django](#) (Django docs) article explains Django's security features and how to secure a Django-powered website.

[Common threats/protections](#)[Edit](#)

Rather than duplicate the Django documentation here, in this article we'll demonstrate just a few of the security features in the context of our Django [LocalLibrary](#) tutorial.

[Cross site scripting \(XSS\)](#)

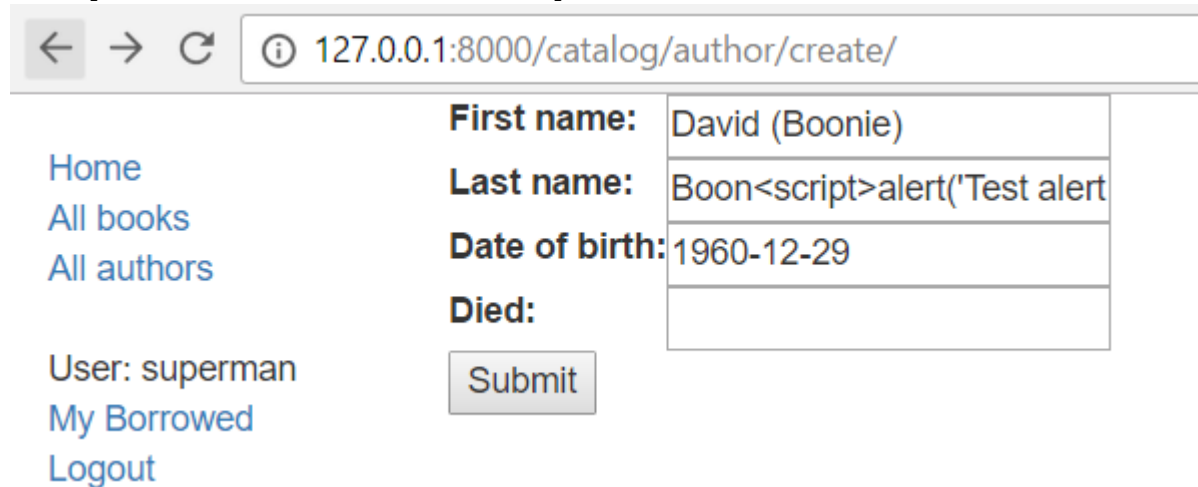
XSS is a term used to describe a class of attacks that allow an attacker to inject client-side scripts *through* the website into the browsers of other users. This is usually achieved by storing

malicious scripts in the database where they can be retrieved and displayed to other users, or by getting users to click a link that will cause the attacker's JavaScript to be executed by the user's browser.

Django's template system protects you against the majority of XSS attacks by [escaping specific characters](#) that are "dangerous" in HTML. We can demonstrate this by attempting to inject some JavaScript into our LocalLibrary website using the Create-author form we set up in [Django Tutorial Part 9: Working with forms](#).

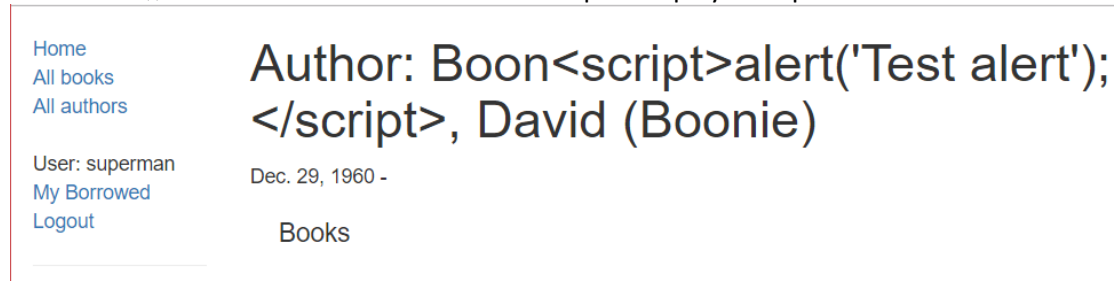
1. Start the website using the development server (`python3 manage.py runserver`).
2. Open the site in your local browser and login to your superuser account.
3. Navigate to the author-creation page (which should be at URL: <http://127.0.0.1:8000/catalog/author/create/>).
4. Enter names and date details for a new user, and then append the following text to the Last Name field:

```
<script>alert('Test alert');</script>.
```



Note: This is a harmless script that, if executed, will display an alert box in your browser. If the alert is displayed when you submit the record then the site is vulnerable to XSS threats.

5. Press **Submit** to save the record.
6. When you save the author it will be displayed as shown below. Because of the XSS protections the `alert()` should not be run. Instead the script is displayed as plain text.



If you view the page HTML source code, you can see that the dangerous characters for the script tags have been turned into their harmless escape code equivalents (e.g. `>` is now `>`;

```
<h1>Author: Boon<script>alert(&#39;Test alert&#39;);</script>;  
David (Boonie) </h1>
```

Using Django templates protects you against the majority of XSS attacks. However it is possible to turn off this protection, and the protection isn't automatically applied to all tags that wouldn't normally be populated by user input (for example, the `help_text` in a form field is usually not user-supplied, so Django doesn't escape those values).

It is also possible for XSS attacks to originate from other untrusted source of data, such as cookies, Web services or uploaded files (whenever the data is not sufficiently sanitized before including in a page). If you're displaying data from these sources, then you may need to add your own sanitisation code.

Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent. For example consider the case where we have a hacker who wants to create additional authors for our LocalLibrary.

Note: Obviously our hacker isn't in this for the money! A more ambitious hacker could use the same approach on other sites to perform much more harmful tasks (e.g. transfer money to their own accounts, etc.)

In order to do this, they might create an HTML file like the one below, which contains an author-creation form (like the one we used in the previous section) that is submitted as soon as the file is loaded. They would then send the file to all the Librarians and suggest that they open the file (it contains some harmless information, honest!). If the file is opened by any logged in librarian, then the form would be submitted with their credentials and a new author would be created.

```
<html>  
<body onload='document.EvilForm.submit()'>  
  
<form action="http://127.0.0.1:8000/catalog/author/create/" method="post"  
name='EvilForm'>  
  <table>  
    <tr><th><label for="id_first_name">First name:</label></th><td><input  
id="id_first_name" maxlength="100" name="first_name" type="text" value="Mad"  
required /></td></tr>  
    <tr><th><label for="id_last_name">Last name:</label></th><td><input  
id="id_last_name" maxlength="100" name="last_name" type="text" value="Man"  
required /></td></tr>  
    <tr><th><label for="id_date_of_birth">Date of  
birth:</label></th><td><input id="id_date_of_birth" name="date_of_birth"  
type="text" /></td></tr>  
    <tr><th><label for="id_date_of_death">Died:</label></th><td><input  
id="id_date_of_death" name="date_of_death" type="text" value="12/10/2016"  
/></td></tr>  
  </table>  
  <input type="submit" value="Submit" />  
</form>  
  
</body>  
</html>
```

Run the development web server, and log in with your superuser account. Copy the text above into a file and then open it in the browser. You should get a CSRF error, because Django has protection against this kind of thing!

The way the protection is enabled is that you include the `{% csrf_token %}` template tag in your form definition. This token is then rendered in your HTML as shown below, with a value that is specific to the user on the current browser.

```
<input type='hidden' name='csrfmiddlewaretoken'
value='0QRWHnYVg776y2l66mcvZqp8alrv4lb8S8lZ4ZJUWGF5VHrVfL2mpH29YZ39PW' />
```

Django generates a user/browser specific key and will reject forms that do not contain the field, or that contain an incorrect field value for the user/browser.

To use this type of attack the hacker now has to discover and include the CSRF key for the specific target user. They also can't use the "scattergun" approach of sending a malicious file to all librarians and hoping that one of them will open it, since the CSRF key is browser specific.

Django's CSRF protection is turned on by default. You should always use the `{% csrf_token %}` template tag in your forms and use `POST` for requests that might change or add data to the database.

Other protections

Django also provides other forms of protection (most of which would be hard or not particularly useful to demonstrate):

SQL injection protection

SQL injection vulnerabilities enable malicious users to execute arbitrary SQL code on a database, allowing data to be accessed, modified, or deleted irrespective of the user's permissions. In almost every case you'll be accessing the database using Django's `QuerySets`/models, so the resulting SQL will be properly escaped by the underlying database driver. If you do need to write raw queries or custom SQL then you'll need to explicitly think about preventing SQL injection.

Clickjacking protection

In this attack a malicious user hijacks clicks meant for a visible top level site and routes them to a hidden page beneath. This technique might be used, for example, to display a legitimate bank site but capture the login credentials in an invisible `<iframe>` controlled by the attacker. Django contains [clickjacking protection](#) in the form of the `X-Frame-Options` middleware which, in a supporting browser, can prevent a site from being rendered inside a frame.

Enforcing SSL/HTTPS

SSL/HTTPS can be enabled on the web server in order to encrypt all traffic between the site and browser, including authentication credentials that would otherwise be sent in plain text (enabling HTTPS is highly recommended). If HTTPS is enabled then Django provides a number of other protections you can use:

- [SECURE_PROXY_SSL_HEADER](#) can be used to check whether content is secure, even if it is incoming from a non-HTTP proxy.
- [SECURE_SSL_REDIRECT](#) is used to redirect all HTTP requests to HTTPS.
- Use [HTTP Strict Transport Security](#) (HSTS). This is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS. Combined with redirecting HTTP requests to HTTPS, this setting ensures that HTTPS is always used after a successful connection has occurred. HSTS may either be configured with [SECURE_HSTS_SECONDS](#) and [SECURE_HSTS_INCLUDE_SUBDOMAINS](#) or on the Web server.
- Use 'secure' cookies by setting [SESSION_COOKIE_SECURE](#) and [CSRF_COOKIE_SECURE](#) to `True`. This will ensure that cookies are only ever sent over HTTPS.

Host header validation

Use [ALLOWED_HOSTS](#) to only accept requests from trusted hosts.

There are many other protections, and caveats to the usage of the above mechanisms. While we hope that this has given you an overview of what Django offers, you should still read the Django security documentation.

[Summary](#)[Edit](#)

Django has effective protections against a number of common threats, including XSS and CSRF attacks. In this article we've demonstrated how those particular threats are handled by Django in our *LocalLibrary* website. We've also provided a brief overview of some of the other protections.

This has been a very brief foray into web security. We strongly recommend that you read [Security in Django](#) to gain a deeper understanding.

The next and final step in this module about Django is to complete the [assessment task](#).

Page url: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/web_application_security

Assessment: DIY Django mini blog

In this assessment you'll use the Django knowledge you've picked up in the [Django Web Framework \(Python\)](#) module to create a very basic blog.

Prerequisites: Before attempting this assessment you should have already worked through all the articles in this module.

Objective: To test comprehension of Django fundamentals, including URL configurations, models, views, forms, and templates.

[Project brief](#)[Edit](#)

The pages that need to be displayed, their URLs, and other requirements, are listed below:

Page	URL	Requirements
Home page	/ and /blog/	An index page describing the site. List of all blog posts: <ul style="list-style-type: none">• Accessible to all users from a sidebar link.• List sorted by post date (newest to oldest).• List paginated in groups of 5 articles.• List items display the blog title, post date, and author.• Blog post names are linked to blog detail pages.• Blogger (author names) are linked to blog author detail pages.
List of all blog posts	/blog/blogs/	Information for a specified author (by id) and list of their blog posts: <ul style="list-style-type: none">• Accessible to all users from author links in blog posts etc.• Contains some biographical information about the blogger/author.• List sorted by post date (newest to oldest).• Not paginated.• List items display just the blog post name and post date.• Blog post names are linked to blog detail pages.
Blog author (blogger) detail page	/blog/blogger/<author-id>	Blog post details. <ul style="list-style-type: none">• Accessible to all users from blog post lists.
Blog post detail page	/blog/<blog-id>	

Page	URL	Requirements
		<ul style="list-style-type: none"> • Page contains the blog post: name, author, post date, and content. • Comments for the blog post should be displayed at bottom. • Comments should be sorted in order: oldest to most recent. • Contains link to add comments at end for logged in users (see Comment form page) • Blog posts and comments need only display plain text. There is no need to support any sort of HTML markup (e.g. links, images, bold/italic, etc).
		List of bloggers on system:
List of all bloggers	<code>/blog/bloggers/</code>	<ul style="list-style-type: none"> • Accessible to all users from site sidebar • Blogger names are linked to Blog author detail pages.
		Create comment for blog post:
Comment form page	<code>/blog/<blog-id>/create</code>	<ul style="list-style-type: none"> • Accessible to logged-in users (only) from link at bottom of blog post detail pages. • Displays form with description for entering comments (post date and blog is not editable). • After a comment has been posted, the page will redirect back to the associated blog post page. • Users cannot edit or delete their posts. • Logged out users will be directed to the login page to log in, before they can add comments. After logging in, they will be redirected back to the blog page they wanted to comment on. • Comment pages should include the name/link to the blogpost being commented on.
		Standard Django authentication pages for logging in, out and setting the password:
User authentication pages	<code>/accounts/<standard urls></code>	<ul style="list-style-type: none"> • Login/out should be accessible via sidebar links.
Admin site	<code>/admin/<standard urls></code>	Admin site should be enabled to allow create/edit/delete of blog posts, blog authors and

blog comments (this is the mechanism for bloggers to create new blog posts):

- Admin site blog posts records should display the list of associated comments inline (below each blog post).
- Comment names in the Admin site are created by truncating the comment description to 75 characters.
- Other types of records can use basic registration.

In addition you should write some basic tests to verify:

- All model fields have the correct label and length.
- All models have the expected object name (e.g. `__str__()` returns the expected value).
- Models have the expected URL for individual Blog and Comment records (e.g. `get_absolute_url()` returns the expected URL).
- The BlogListView (all-blog page) is accessible at the expected location (e.g. `/blog/blogs`)
- The BlogListView (all-blog page) is accessible at the expected named url (e.g. `'blogs'`)
- The BlogListView (all-blog page) uses the expected template (e.g. the default)
- The BlogListView paginates records by 5 (at least on the first page)

Note: There are of course many other tests you can run. Use your discretion, but we'll expect you to do at least the tests above.

The following section shows [screenshots](#) of a site that implements the requirements above.

[Screenshots](#)[Edit](#)

The following screenshot provide an example of what the finished program should output.

[List of all blog posts](#)

This displays the list of all blog posts (accessible from the "All blogs" link in the sidebar). Things to note:

- The sidebar also lists the logged in user.
- Individual blog posts and bloggers are accessible as links in the page.
- Pagination is enabled (in groups of 5)
- Ordering is newest to oldest.

[←](#) [→](#) [↻](#) [127.0.0.1:8000/blog/all/](#) [☆](#) [⋮](#)

[Home](#)
[All blogs](#)
[All bloggers](#)

User: ubuntu
[Logout](#)

All blogs

- [Why can't dogs get along with cats?](#) (Nov. 7, 2016) — [dr_evil](#)
- [Coffee drinkers of the world unite](#) (Nov. 7, 2016) — [dr_evil](#)
- [When is it OK to eat snails?](#) (Nov. 6, 2016) — [fishman](#)
- [The best movie of all time?](#) (Nov. 1, 2016) — [johncitizen](#)
- [Eliminating your rivals 101](#) (Oct. 12, 2016) — [dr_evil](#)

Page 1 of 2. [next](#)

List of all bloggers

This provides links to all bloggers, as linked from the "All bloggers" link in the sidebar. In this case we can see from the sidebar that no user is logged in.

[←](#) [→](#) [↻](#) [127.0.0.1:8000/blog/bloggers/](#)

[Home](#)
[All blogs](#)
[All bloggers](#)

[Login](#)

All bloggers

- [ubuntu](#)
- [fishman](#)
- [dr_evil](#)
- [johncitizen](#)

Blog detail page

This shows the detail page for a particular blog.

[Home](#)
[All blogs](#)
[All bloggers](#)

User: ubuntu
[Logout](#)

Why can't dogs get along with cats?

Post date Nov. 7, 2016

Author: [dr_evil](#)

Description:

Since the dawn of time, cats and dogs have been enemies. Dogs have hated cats, and cats have feared and loathed their canine foes.

The reasons for this can be traced back to the early days, when furosaur first walked the earth in the cretaceous period. It all started when the first dog precursor (fangasaurus) met the first ...

Comments

fishman (Nov. 7, 2016, 1:42 a.m.) - This sounds like very poor science. Where did you do your research?

dr_evil (Nov. 7, 2016, 2:05 a.m.) - This is my blog, I can write whatever I like.

[Add a new comment](#)

Note that the comments have a date *and* time, and are ordered from oldest to newest (opposite of blog ordering). At the end we have a link for accessing the form to add a new comment. If a user is not logged in we'd instead see a suggestion to log in.

dr_evil (Nov. 7, 2016, 2:05 a.m.) - This is my blog, I can write whatever I like.

[Login to add a new comment](#)

[Add comment form](#)

This is the form to add comments. Note that we're logged in. When this succeeds we should be taken back to the associated blog post page.

[Home](#)
[All blogs](#)
[All bloggers](#)

User: ubuntu
[Logout](#)

Post your comment for: [Why can't dogs get along with cats?](#)

Description:

I don't think it helps to blame

Enter comment about blog here.

Submit

Author bio

This displays bio information for a blogger along with their blog posts list.

[Home](#)
[All blogs](#)
[All bloggers](#)

[Login](#)

Blogger: dr_evil

Bio

According to Wikipedia: Dr. Evil is a fictional character, played by Mike Myers in the Austin Powers film series. He is the antagonist of the movies, and Austin Powers' nemesis. He is a parody of James Bond villains, primarily Donald Pleasence's Ernst Stavro Blofeld (as featured in You Only Live Twice).

Blogs list

- [Why can't dogs get along with cats?](#) (Nov. 7, 2016)
- [Coffee drinkers of the world unite](#) (Nov. 7, 2016)
- [Eliminating your rivals 101](#) (Oct. 12, 2016)

Steps to complete [Edit](#)

The following sections describe what you need to do.

1. Create a skeleton project and web application for the site (as described in [Django Tutorial Part 2: Creating a skeleton website](#)). You might use 'diyblog' for the project name and 'blog' for the application name.
2. Create models for the Blog posts, Comments, and any other objects needed. When thinking about your design, remember:
 - Each comment will have only one blog, but a blog may have many comments.
 - Blog posts and comments must be sorted by post date.
 - Not every user will necessarily be a blog author though any user may be a commenter.
 - Blog authors must also include bio information.

3. Run migrations for your new models and create a superuser.
4. Use the admin site to create some example blog posts and blog comments.
5. Create views, templates, and URL configurations for blog post and blogger list pages.
6. Create views, templates, and URL configurations for blog post and blogger detail pages.
7. Create a page with a form for adding new comments (remember to make this only available to logged in users!)

Hints and tips [Edit](#)

This project is very similar to the [LocalLibrary](#) tutorial. You will be able to set up the skeleton, user login/logout behaviour, support for static files, views, URLs, forms, base templates and admin site configuration using almost all the same approaches.

Some general hints:

1. The index page can be implemented as a basic function view and template (just like for the `locallibrary`).
2. The list view for blog posts and bloggers, and the detail view for blog posts can be created using the [generic list and detail views](#).
3. The list of blog posts for a particular author can be created by using a generic list `Blog list view` and filtering for blog object that match the specified author.
 - You will have to implement `get_queryset(self)` to do the filtering (much like in our library class `LoanedBooksAllListView`) and get the author information from the URL.
 - You will also need to pass the name of the author to the page in the context. To do this in a class-based view you need to implement `get_context_data()` (discussed below).
4. The *add comment* form can be created using a function-based view (and associated model and form) or using a generic `CreateView`. If you use a `CreateView` (recommended) then:
 - You will also need to pass the name of the blog post to the comment page in the context (implement `get_context_data()` as discussed below).
 - The form should only display the comment "description" for user entry (date and associated blog post should not be editable). Since they won't be in the form itself, your code will need to set the comment's author in the `form_valid()` function so it can be saved into the model ([as described here](#) — Django docs). In that same function we set the associated blog. A possible implementation is shown below (`pk` is a blog id passed in from the URL/URL configuration).

```

def form_valid(self, form):
    """
    Add author and associated blog to form data before
    setting it as valid (so it is saved to model)
    """
    #Add logged-in user as author of comment
    form.instance.author = self.request.user
    #Associate comment with blog based on passed id
    form.instance.blog=get_object_or_404(Blog, pk =
self.kwargs['pk'])
    # Call super-class form validation behaviour
    return super(BlogCommentCreate, self).form_valid(form)

```

and "reverse" the URL for the original blog. You can get the required blog ID using the `self.kwargs` attribute, as shown in the `form_valid()` method above.

We briefly talked about passing a context to the template in a class-based view in the [Django Tutorial Part 6: Generic list and detail views](#) topic. To do this you need to override `get_queryset()` (first getting the existing context, updating it with whatever additional variables you want to pass to the template, and then returning the updated context. For example, the code fragment below shows how you can add a blogger object to the context based on their `BlogAuthor` id.

```
class SomeView(generic.ListView):
    ...

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(SomeView, self).get_context_data(**kwargs)
        # Get the blogger object from the "pk" URL parameter and add it to
the context
        context['blogger'] = get_object_or_404(BlogAuthor, pk =
self.kwargs['pk'])
        return context
```

[Assessment](#)[Edit](#)

The assessment for this task is [available on Github here](#). This assessment is primarily based on how well your application meets the requirements we listed above, though there are some parts of the assessment that check your code uses appropriate models, and that you have written at least some test code. When you're done, you can check out our [the finished example](#) which reflects a "full marks" project.

Once you've completed this module you've also finished all the MDN content for learning basic Django server-side website programming! We hope you enjoyed this module and feel you have a good grasp of the basics!

Page url: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/django_assessment_blog