Table of Contents

[Table of Contents 1](#_Toc533975842)

[Preparations 2](#_Toc533975843)

[Starting the project 8](#_Toc533975844)

[URLS intro 12](#_Toc533975845)

[Models intro 19](#_Toc533975846)

[Django Admin site 37](#_Toc533975847)

[Home Page 57](#_Toc533975848)

[Generic list and detail views 75](#_Toc533975849)

[Sessions 101](#_Toc533975850)

[User Authentication and Permissions 108](#_Toc533975851)

[Working with forms 155](#_Toc533975852)

[Testing a Django Application 188](#_Toc533975853)

# Preparations

open terminal, her we put commands and we get replies.

First check python version using the code below, then press enter.

python3 -V

should output something like Python 3.6.5

here we are using the command called python, and we are using an option or argument called -V that stands for version.

if you did not have (which I'm sure you do!) we can install this.

there is a program called apt. copy paste all the lines below and press enter. it will ask you to enter the password, the termnal will not display anything on the screen while you are typing your password, just press enter after you are done.

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt-get update

sudo apt-get install python3.6

after we have python we need to use a tool within python called pip. pip installs utilities and libraries that python needs. we use apt again to install pip. type this and press enter:

sudo apt install python3-pip

We need virtualenvironements to run our scripts (essentially isolated sand boxes)

the library that provides this for us is called virtualenvwrapper

we need to download this using pip:

sudo pip3 install virtualenvwrapper

a good code editor is atom (very natural)

to see if we have it we just run

atom

if this didn't open atom for us, it means that we do not have it, so lets download it using:

sudo add-apt-repository ppa:webupd8team/atom

sudo apt-get update

sudo apt-get install atom

sudo before any command means we are running it as admin (it will ask for password after you enter, you need to type in you password)

apt is a tool that helps us install or remove things. add repository will add a shelf for apt to pick up a package that we wish to install.

apt-get update will update the things we have on the shelf we can reach. apt-get install will grab something from the shelf and installs it.

After it is installed there are 2 ways to use this editor. Either just type in atom to open it, write something then save it. Or when you are in a directory that contains a text file, (eg. test.txt) then we can open it using atom, edit it, and then save it and close atom.

since we do not have anything yet, lets open atom and write something.

atom

write something in a new file, save it (lets say as test.txt)and exit.

Back in terminal write:

ls

ls will list stuff in the current directory

now you will see test.txt

if you want to open it, type in:

atom test.txt

now you can edit it and leave again.

to delete it type:

rm test.txt

now type in

ls

and you will not see the file anymore.

to make a directory type in

mkdir newdir

now try

ls

you see the new directory

to go inside it type

cd newdir

you will observe a change in the green shell text to the left. it now has a new blue part called ~/newdir showing we are in that directory.

to print our current working directory try:

pwd

this will show something like /home/user/newdir

to go back to where we were type in:

cd ..

the computer under unix (linux) architecture is essentially a series of files and directories.

type in

cd ~

from no matter where you are to make us go back to home/user (the users main folder)

and type in:

cd /

to go the very root of the computer

then if we want to go back either try cd ~ or we can do:

cd home/user/

where user is the name of the user you have chosen.

I very much suggest you go through this tutorial on web and test it our on terminal before going any further: https://www.guru99.com/terminal-file-manager.html

on top of commands we can add options to it, for example the command ls (the command to list current directory) can be used with extra options such as

ls -l

this will also show details such as date a file was added, read write execute permissions and other things.

there is another option:

ls -a

this will show hidden files

or you can do the two options in conjunction such as:

ls -la

another thing we must know before going any further is variables. we have them in programming scripts. but also we have them in terminal sessions. every time you open the terminal we are in a session. Sometimes we have variables during a session, sometimes we have variables that are by default and will be set in all sessions. for example try:

myvar='a'

this will make a new variable called myvar and the value of it is a character a.

to see it works try:

echo $myvar

this will output 'a' as required. but the problem is that this will not be saved when we close terminal and open it again. so we need to save this somewhere. this is called .bashrc, the place where we store our preferences and variables and other things. note that anything with . at the beginning is a hidden file. to see it we need to use ls -a as explained before.

anyway back to where we were, go back to the current user folder using:

cd ~

then open the hidden file .bashrc this file saves all your preferences in the terminal as well as certain variables, where things should be and so on.

to even know where it is use:

ls -a

we can see it listed

then use:

atom .bashrc

to open it up in atom

add the following text at the bottom and save it.

export WORKON\_HOME=$HOME/.virtualenvs

export VIRTUALENVWRAPPER\_PYTHON=/usr/bin/python3

export VIRTUALENVWRAPPER\_VIRTUALENV\_ARGS=' -p /usr/bin/python3 '

export PROJECT\_HOME=$HOME/Devel

source /usr/local/bin/virtualenvwrapper.sh

so export will create a variable and save it. for example the first line will create a variable called WORKON\_HOME and set it to variable $HOME+/.virtualenvs

first lets just check what HOME is. try:

echo $HOME

to see what home is. it is the users main directory. so export WORKON\_HOME=$HOME/.virtualenv means we want to have a varible called WORKON HOME that is pointing to .virtualenv file in our directory.

similarly the second line: export VIRTUALENVWRAPPER\_PYTHON=/usr/bin/python3 means that we want our virtualenvwrapper variable to be where our python is.

python is a program, programs are files. then our python is as the previous line hints, exists in root folder, under usr folder then in bin folder.

to check if it is really there, try:

ls /usr/bin/

and you must find python3 in there.

the last line: source /usr/local/bin/virtualenvwrapper.sh means we want to access this script under the mentioned folder. the script virtualenvwrapper.sh is essentially what helps us to have this sandbox or virtual environment for our project.

now we need to reload. by this we mean that we want to run the startup file again (.bashrc) to do so:

source .bashrc

this will show some lines showing certain things being created for us.

now let us create a new virtual environment, lets call it env:

mkvirtualenv env

note that now we can see the (env) at the beginning of the line showing we are in our sandbox. whatever we do now will not effect the rest of our computer essentially.

lets install our django. django is a tool helping us with web developement. do this by:

pip3 install django

to test this is successful we shall try to get the version of django we have using:

python3 -m django --version

this will say something like 2.1.4

lets now try it out, first make a folder:

mkdir django\_test

note that it is forbidden to have spaces in any file or folder name, so use \_ if you must.

then navigate to this folder using:

cd django\_test

quick tip, terminal has auto completion. to use this feature first go back using

cd ..

then type

cd dj

before you press enter, press tab button. this will autocomple it. if it doesn’t do so, try pressing tab twice in a row. this will show all possible auto completion.

anyway when you are inside the folder type in

django-admin startproject mytestsite

cd mytestsite

this will create a folder called mytestsite, within it we have another folder of the same name, this has some utility file we talk about later, as well as a a file called manage.py

run this file and tell it to run the server using the below code:

python3 manage.py runserver

there will be some errors but don't worry about them at this stage. essentially we have a bare skeleton server running. we can add to this a website and django will run it for us. lets just visit the website that django is running currently, click on the url

http://127.0.0.1:8000/

seen before the last line.

you can either just click on it, or copy paste it using control+shift+C and then paste in in browser such as firefox and running it.

to quit (stop the server), do control+C

to leave the virtualenv try:

deactivate

you shall see that the (env) from the beginning of the line has gone now.

to use the environment again, use:

workon env

# Starting the project

Okey we want to create a library website, we want users to have accounts, search for books, maybe reserve books and other things.

In computing we have applications that are essentially software that do a certain job. for example Microsoft words is an offline application that edits text. amazon or google are online applications. any application actually resides on a physical place.

For example the words application is saved somewhere within your computers storage space. and when you want to use it, it is running on memory and shown to you. similarly a web application is actually physically saved somewhere. for example google has humongous data stores around the world and even in space, these data stores are loaded on to a server and are running while google website it up and running. similarly our website is an application. it needs to be saved on a folder in our computer. then using django we can run it (server will be running) and then we can access it via a browser.

so lets call this folder locallibrary. this folder must have another subfolder with the same name (locallibrary) that has some project files in it. as well as another folder called catalog for example which is the name of the application that we want to make, and the users use this application in real life. shown below:

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)

lets make the folder first and navigate inside it:

mkdir django\_projects

cd django\_projects

then make the locallibrary folder using django and let it create all necessary tools for us. then go inside the subfolder with the same name:

django-admin startproject locallibrary

cd locallibrary

the structure is as shown below: (note that the / symbol eg. locallibrary/ signifies that the there is a folder called locallibrary, inside which we find files and or folders. here we see there is a file called manage.py inside locallibrary folder, also another locallibrary folder within the first one. inside the inner locallibrary folder we find an initialiser script called \_\_init\_\_.py, a settings script called settings.py, a urls.py that has our current urls that the user will open and use in his or her browser, and finally wsgi.py which helps django application to talk to the web server.)

locallibrary/

  manage.py

  locallibrary/

\_\_init\_\_.py

    settings.py

    urls.py

    wsgi.py

to see the current working directory use the below command:

pwd

this should say something like:

/home/user/django\_project/locallibrary

next lets actually make our catalog application using:

python3 manage.py startapp catalog

now our directory will look like:

locallibrary/

  manage.py

  locallibrary/

**catalog/**

**admin.py**

**apps.py**

**models.py**

**tests.py**

**views.py**

**\_\_init\_\_.py**

**migrations/**

views holds what the user views or sees in the browser, model has models as the name suggests, tests has the tests again as the name suggests, admin has administration site configurations and apps holds the application registration. let us not worry too much about remembering these, we will use it when it comes to it.

migrations folder holds files that allow us to automatically update our database as we modify our models. think about models as excel files that have records on books and our data to be the entries inside these excel files. a database is the collection of these files sometimes called models or tables.

\_\_init\_\_.py again is a empty file called an initialiser. why do we need it? it is more or less formality but essentially it tells python that this folder (catalog) is a package that we can use. what is a package? when we write a script, for example test.py, maybe to run this we need a package and we use some of the functionality of this package. this process is called importing a package within a script. so without \_\_init\_\_.py we cannot import content of this folder to a script outside this folder.

there is something missing from catalog. a place to put our urls (web pages a user can access), templates (essentially templates for a web page) and static files (like images). we shall do this soon.

next we need to register this new app in our settings.py in the main folder. think of this as google providing a new service, like google maps, but not having anyway to access it. so we need to say that now our locallibrary website has a new app called catalog. to do this open the settings file located in:

django\_projects/locallibrary/locallibrary/settings.py

using:

atom localibrary/settings.py

(note we are currently in the main locallibray file, so we are opening settings.py file which is located under the sub folder locallibrary.)

in this file find a section called installed apps. and modify it by appending the line 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',**

]

this will tell django at the main folder that we have the new app going on.

there are many important things in this file. firstly the installed apps, then the middleware which have functionality like sessions and authentication (we touch on this later, sessions are created when you log in to a website with username and password, and you are kept logged in, meaning you are in an active session, authentication refers to how a user is authenticated in our website) another important thing here is the database settings.

a database is a storage that stores data. it has an engine as it needs to retrieve or store data actively. so scroll to find the database section. note the engine is set to sqlite3 which is a simple and lightweigh engine (compared to mysql or oracle), we have a name for this database which is called 'locallibrary.sqlite3' as our base directory is called locallibrary.

we can also have other options here (dont add it! just for demonstration purposes):

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mydatabase',

'USER': 'mydatabaseuser',

'PASSWORD': 'mypassword',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

this database for example uses a more advanced engine called postgresql, the name is mydatabase, there is only one user that can use this database (many actual real life users can be logged in using this username to access the database however.), the user has a password to be authenticated with, there is also a setting called host, (this number is essentially the IP address, like a licence plate, of the computer that is connected to internet network and is sharing this database and the relevant application ) and finally a port (a computer has many ports, for example when you send an email to someone, this email travels down a different port and tunnel in the internet as opposed to when we click on a link in the web, our request to see a link is travelled down a different port. all these port numbers have special meaning, but lets not worry about it now).

lets just change another thing in the settings file. go to timezone and change it to:

TIME\_ZONE = 'Europe/London'

lets save and exit. in terminal make sure we are in locallibrary (the main folder not the subfolder) lets call this folder from now on our project folder.

# URLS intro

here lets open up the file called urls.py under the subfolder locallibrary. so use:

atom localibrary/urls.py

again this file is actually located in:

locallibrary/locallibrary/urls.py

Why do we need this? users of the website will go to urls and from there they need to access our applications. but our apps are in python so we need a python module (a module is just a script) that maps urls to application (or rather views).

this module is called **URLconf** (url configuration). it is written in python. this mapping can even reference other mappings. it can even be constructed dynamically (we don’t need to specifically write every bit of it).

Lets see how this works in real life. suppose we have the below in our urlconf file:

**from** **django.urls** **import** path

**from** **.** **import** views

urlpatterns = [

path('articles/2003/', views.special\_case\_2003),

path('articles/<int:year>/', views.year\_archive),

path('articles/<int:year>/<int:month>/', views.month\_archive),

path('articles/<int:year>/<int:month>/<slug:slug>/', views.article\_detail),

]

and there is a user typing this in the browser:

**www.websitename.com/articles/2005/03/**

the browser then makes an object called 'request'. think of this as an order in a restaurant made by your waiter. then the order is given to django (the chef!).

then django will start opening the urlconf file (this is actually the default urlconf file called root\_urlconf, there can be more than one, so unless specified we open the default one).

django will then find the section shown above (urlpatterns) then he decides which one to pick. in this case the third one is selected. why? <int;year> is a variable part of the url, so 2003 will match that, then similarly <int;month> matchs the 03 part of the url.

then django will execute the function associated with this order or request, which in this case is views.month\_archive. this function as name implies is inside the views.py file. we have imported this file as seen above using 'from . import views' (read it like from root import views file and access all functions within it.).

then this function will be executed with an instance of the order or request itself, as well as the year and the month like below:

**views.month\_archive(request, year=2005, month=3)**.

another example

**www.websitename.com/articles/2003**

will actually not match anything as there is no / at the end of it. so django will display an error.

anyway when we open our own urls.py (which is our urlconf) file we see something like this: (remember this file is inside the locallibrary subfolder, so the path to it is: /home/urser/djangoproject/locallibrary/locallibrary/urls.py)

to open it either navigate to go inside the subfolder using (cd command) or just type

atom home/user/djangoproject/locallibrary/locallibrary/urls.py

to get:

"""locallibrary URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:

    https://docs.djangoproject.com/en/2.1/topics/http/urls/

Examples:

Function views

    1. Add an import:  from my\_app import views

    2. Add a URL to urlpatterns:  path('', views.home, name='home')

Class-based views

    1. Add an import:  from other\_app.views import Home

    2. Add a URL to urlpatterns:  path('', Home.as\_view(), name='home')

Including another URLconf

    1. Import the include() function: from django.urls import include, path

    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))

"""

from django.contrib import admin

from django.urls import path

urlpatterns = [

    path('admin/', admin.site.urls),

]

urlpatterns (this is a list in python, the objects inside of it are of type path() ) each path object associates a URL to a specific view as described before) is at the bottom.

we see that there is already one url and application pair in our urlpatterns. so currently the url that begins with admin/ will be mapped to the module 'admin.site.urls'.

lets test this theory out. make sure we are in our virtual environment first. by

workon env

then do:

python3 manage.py runserver

then open the url displayed at the bottom of the result, (the part where it says starting developement at http://...)

open this up in the browser. then add /admin/ at the end of the url that is opened. so on top of the browser (in the bookmark bar) it should say: 127.0.0.1:800/admin/

and it should open a login page for admin, don’t worry about logging in now, we just wanted to see how this works.

so go back to terminal, do ctrl+C and end running it.

now go back to atom where we had the urlconf file open. add the lines below to it: (\*\*\*)

# Use include() to add paths from the catalog application

from django.urls import include

from django.urls import path

urlpatterns += [

    path('catalog/', include('catalog.urls')),

]

what does it do? anytime the user wants to go to

127.0.0.1:8000/catalog

django will forward the request to catalog.urls module. (this is the view or app that we made for the catalog functionality.)

so we saw that user when he tries to go to the url:

127.0.0.1:8000

or our root url (this is denoted by '' or empty single quotes), he goes to the main page with the django rocket launching. but we want the user to be instead redirected to our catalog application.

add this to the bottom of urls.py file:

#Add URL maps to redirect the base URL to our application

from django.views.generic import RedirectView

urlpatterns += [

    path('', RedirectView.as\_view(url='/catalog/', permanent=True)),

]

what is this? an application must have its url as discussed before so we had this url for catalog application:

27.0.0.1:8000/catalog/

we need to redirect the user using a function called RedirectView. like any function we could have inputs and outputs. we see that this function takes two inputs inside the () openining to the right of its name, one of them is url, the second is permanent. generally this format goes for python functions.

**FUNCTIONAME(INPUT1NAME=valueofinput1, INPUT2NAME=valueofinput2, ...)**

url's value is set to '/catalog/' and permanent's value is set to True. meaning we want the root url '' to the left of the function name to be redirected to '/catalog/' and we want this change to be permanent.

essentially we have added a new path() object that takes '' and outputs '/catalog/' permanently.

Another thing we want to add here is serving static files. so far we have only seen what will happen if the user requests a URL that gets mapped to an application and then that application is served to user. but what about static files like images or CSS or Javascript?

add the following lines to the bottom of the urls.py file

# 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)

this will enable us to serve static files during developement.

essentially our urls pattern is going to be as follows (we did it in three separate parts using urlspattern += ...)

urlpatterns = [

  path('admin/', admin.site.urls),

  path('catalog/', include('catalog.urls')),

path('', RedirectView.as\_view(url='/catalog/', permanent=True)),

] + static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT)

this will technically do the same job as what we have.

my urls.py file inside my library subfolder (~/django\_project/locallibray/ locallibray/urls.py) is now looking like this:

|  |  |
| --- | --- |
| """locallibrary URL Configuration |  |
|  |  |
|  | The `urlpatterns` list routes URLs to views. For more information please see: |
|  | https://docs.djangoproject.com/en/2.1/topics/http/urls/ |
|  | Examples: |
|  | Function views |
|  | 1. Add an import: from my\_app import views |
|  | 2. Add a URL to urlpatterns: path('', views.home, name='home') |
|  | Class-based views |
|  | 1. Add an import: from other\_app.views import Home |
|  | 2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home') |
|  | Including another URLconf |
|  | 1. Import the include() function: from django.urls import include, path |
|  | 2. Add a URL to urlpatterns: path('blog/', include('blog.urls')) |
|  | """ |
|  | from django.contrib import admin |
|  | from django.urls import path |
|  |  |
|  | urlpatterns = [ |
|  | path('admin/', admin.site.urls), |
|  | ] |
|  |  |
|  | #Use include() to add paths from catalog application |
|  |  |
|  | from django.urls import include |
|  | from django.urls import path |
|  |  |
|  | urlpatterns+=[ |
|  | path('catalog/', include('catalog.urls')), |
|  | ] |
|  |  |
|  | #Add URL maps to redirect the base URL to our application |
|  | from django.views.generic import RedirectView |
|  | urlpatterns+=[ |
|  | path('',RedirectView.as\_view(url='/catalog/',permanent=True)), |
|  | ] |
|  |  |
|  | #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) |

and finally we need to actually have a urls.py file for our catalog. why? scroll up to (\*\*\*), we have entered a code inside urls.py file inside locallibrary. this code reads:

path('catalog/', include('catalog.urls')),

so when user types in catalog/ at the end of the url, we would want to match that with something from catalog folder. catalog.urls means we want a urls file inside catalog folder.

so lets make this file. first make sure we are either in catalog folder in terminal (do so via cd command) or simply type this from anywhere (as we are mentioning the path completely not relatively):

atom ~/django\_project/locallibrary/catalog/urls.py

this will open up atom with a new file. copy paste this content to it:

from django.urls import path

from . import views

urlpatterns = [

]

and save it (making sure it is saved at ~/django\_project/locallibrary/catalog/).

now lets test out website. go to terminal, make sure we are in the environment 'env'. then make sure we are in the locallibrary main folder. type:

python3 manage.py runserver

again copy the link, go to browser and run it (we actually get an error, this is because we do not actually have any pages or urls defined in the catalog.urls module, as you see above this is literally empty [] ).

end it with ctrl+C.

# Models intro

in our application we have models. they logically model our data. for example since we are going to have to store data on books, we need to define a model for book. a model essentially defines an object. in this model we need to have things like book.name , book.publicationdate, book.author and etc. they are called attributes of the model book.

so in models.py we will be making such definitions of models.

however this is not a database. this is just model of what kind of object we will have. a database actually holds the data for each type of object. we can create a database, or we can let django do it for us.

so we say what a book is, what a user is ... (logically), we store these in models.py. and django will construct a database based on these and then we can store details of books inside it.

as we change our model, django should change the database too. for example suppose all of the sudden we need to store a book.publisher attribute for every book. this is a logical change in models. then we need to have something called database migrations scripts that helps track the changes on models and make relevant changes on database.

lets first go to the main locallibrary directory (where manage.py is located) using cd command. then type in:

python3 manage.py makemigrations

python3 manage.py migrate

we first create the migrations for all application in this folder (in this case catalog) then we actually apply migrations to our database.

My explanation of migration is very basic if I am honest here, if you would like to know more try visiting this link: https://docs.djangoproject.com/en/2.1/topics/migrations/

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.

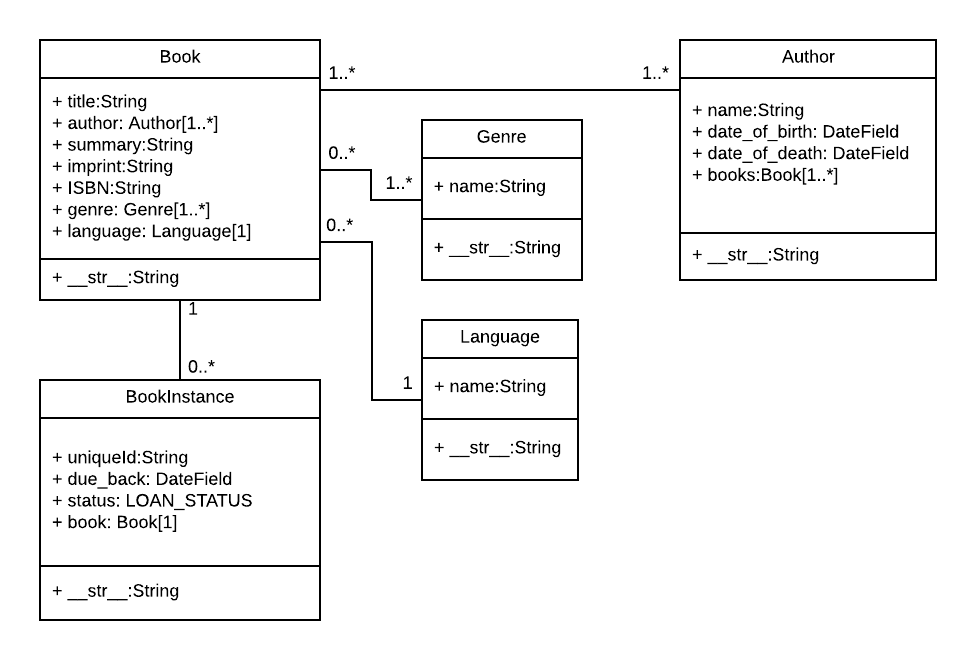
When designing your models it makes sense to have separate models for every "object" (group of related information). In this case the obvious objects are books, book instances and authors.

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 of course since you are a translator (English, French, Japanese).

Another thing to decide is relationships between different type of objects. For example imagine we have already made a model for 'author' and another for 'book' (note they are all single, we never have a model name plural). What is the relationship between book and author? An author wrote many books, but a book has only one author, we call this **MANY\_TO\_ONE.** Now think about the relationship between a book and an ISBN, clearly this relationship is **ONE\_TO\_ONE.** And finally think about an author and a publisher, the author can have contracts with many publishers and a publisher can have many authors writing books for them, so **MANY\_TO\_MANY.**  These are more or less all the relationships we need to consider.

In Django one to one is called (OneToOneField), one to many is (ForeignKey) and many to many is (ManyToManyField).

With that in mind, the UML (unified modelling language) 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 (recall the page we could reach when we typed in the browser the address of our website followed by /admin, that is the admin interface).



for a more in depth introduction to UML diagrams look at: https://www.eetimes.com/document.asp?doc\_id=1255046

Couple of things to note in the diagram. First from the point of view of the database, each rectangle (model) is called a table. Think of it as an excel file where we put records of objects of that type. So for example 'book' is a long list of various different books (along with all their details such as title, author, ...).

From the point of view of the python application, each rectangle is a function (a method or procedure, they have many different names) in the models.py script. A function is in real life a single book, so we need as many instances of the 'book' function as the number of books we have in our library. So the code we write in our models.py file is a pallet from which we instantiate many instances of book.

Inside each rectangle of a function we have a name on top, the attributes (fields) in the middle and the return on the bottom.

Like any other function they have some kind of output. the outputs for all of the above models are +\_\_str\_\_:String. This signifies they return a string of all the data they have inside it.

Fields have data types. For example book has a field title. Title of a book is for example "Little Prince". So it makes sense for us to store this value (of all book.titles) as a String in python. Similarly look at the author function, ther we have a field called date\_of\_birth. It only makes sense for us to store this under a DateField datatype in python.

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. So the other alternative is to have a separate rectangle for Book status, which only has one field Loan\_status, which is either yes or no. It makes more sense to merge it with Book\_Instance. Essentially in Book we store all unique books we have. For each actual book that we have we have an instance of Book\_instance, so clearly a there can exist many book instances which are the same book object (we have 30 copies of Gordon Ramsies cook books for example! so we need only one object of type Book for this book but 30 instances of BookInstance to store them all).

Also note the lines between every rectange. the 1..\* denotes one to many, 0..\* denotes zero to many and 1 denotes one and only one. For example look at Book and Language. Every book is in one and only one language and every language is seen in 0 or more books.

Within each of the boxes you can see the model name, the field names and types, and also the methods and their return types.

For a more in-depth introduction to databases and the notations above I highly encourage this tutorial written by UCL: https://www.ucl.ac.uk/archaeology/cisp/database/manual/node1.html

Models in Django are under the models.py file. They are implemented (based on) a library in django: django.db.models.Model (note that this is a phisycal thing stored in your computer, this library is somewhere in your usr folder, then it is in a folder called django, then in a folder called db, and so on, so look at the dots in the name as the folders within folders).

Models in django can have fields, methods (they can do something and return an output), and metadata (data about their own data). The code below shows a model (just for demonstration purposes), this model is called 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.my\_field\_name

Fields:

A model can have an arbitrary number of fields, of any data type. in excel these represent the columns of a the actual file, so each entry has a value in the column title, year and so on.

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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#help-text): Provides a text label for HTML forms (e.g. in the admin site), as described above.
* [verbose\_name](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#field-options).

Common field types

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

* [CharField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.TextField) is used for large arbitrary-length strings. You may specify a max\_lengthfor the field, but this is used only when the field is displayed in forms (it is not enforced at the database level).
* [IntegerField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#_blank) is a field for storing integer (whole number) values, and for validating entered values as integers in forms.
* [DateField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#datefield) and [DateTimeField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#datetimefield) are used for storing/representing dates and date/time information (as Python datetime.date in and datetime.datetimeobjects, 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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#emailfield) is used to store and validate email addresses.
* [FileField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#filefield) and [ImageField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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](https://docs.djangoproject.com/en/2.1/ref/models/fields/#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.). A full list is found at: https://docs.djangoproject.com/en/2.1/ref/models/fields/#field-types

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: https://docs.djangoproject.com/en/2.1/ref/models/options/

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 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. As seen in the example above the method returns field name:

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)])

So for example we use the url /mywebsite/catalog/2 to display the book details of the book with id 2. 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 a URL of the right format.

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/inputs).

Aside from defining a model there is another operation called querying models. Think of the section above as building a restaurant where food is stored, and querying as customers ordering food.

Querying can again be divided in to separate actions. Once we might want to get all data in our database (ordering all meals in a restaurant at once), we might want part of the database which is defined by a certain characteristic (in a restaurant ordering all meals with less than 500 calories), we might want to modify an existing record(s) or deleting existing record(s), and finally adding records.

Creating and modifying records:

To create a record you can define an instance of the model and then call save(). In our library think of this as walking in with a new book and storing it in the library, so creating a new record in the database.

# Create a new record using the model's constructor.

record = MyModelName(my\_field\_name="Instance #1")

# Save the object into the database.

record.save()

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. Usually the id is auto incremented, so the next record will have the id 2.

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

# Access model field values using Python attributes.

print(record.id) # should return 1 for the first record.

print(record.my\_field\_name) # should print 'Instance #1'

# Change record by modifying the fields, then calling save().

record.my\_field\_name = "New Instance Name"

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). Clearly if no criteria is given then all records will be handed out.

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()

For example the above statement gets all the books, then we can do a for loop over (to refresh your memory on for loops I suggest this website: https://www.w3schools.com/python/python\_for\_loops.asp).

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.

have this as the full list: https://docs.djangoproject.com/en/2.1/ref/models/querysets/#field-lookups

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:

# Will match on: Fiction, Science fiction, non-fiction etc.

books\_containing\_genre = Book.objects.filter(genre**\_\_**name**\_\_**icontains='fiction')

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 https://docs.djangoproject.com/en/2.1/topics/db/queries/

Now lets work on our library, first open models.py this is located at .../locallibrary/catalog so inside the catalog application folder.

open this using atom

atom models.py

we will find:

from django.db import models

# Create your models here.

essentially so far we have the models module imported from django db. This contains the base class which all our models will inherit from and will be based on.

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."""

    name = models.CharField(max\_length=200, help\_text='Enter a book genre (e.g. Science Fiction)')

    def \_\_str\_\_(self):

        """String for representing the Model object."""

        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.

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)

# 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

    author = models.ForeignKey('Author', on\_delete=models.SET\_NULL, null=True)

    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>')

    # 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.

genre = models.ManyToManyField(Genre, help\_text='Select a genre for this book')

    def \_\_str\_\_(self):

        """String for representing the Model object."""

        return self.title

    def get\_absolute\_url(self):

        """Returns the url to access a detail record for this book."""

        return reverse('book-detail', args=[str(self.id)])

If I have not mentioned note that any segment of code preceded by a # is a comment and will not be looked at by python (computer will skip it and it only serves as a guide for the coder), any code in the triple double quotes """ text """ is a comment that is in multiple lines.

the last line, return reverse('book-detail', args=[str(self.id)]) maybe needs a little bit more explanation.

In the previous chapter we saw that there is a url-mapper functionality which receives a url for example websitename/catalog/2 (the url to go and see the book with id 2). and then returns the actual function that shows this book.

the reverse of this url mapper will get the actual book and returns the url for it. so we are in need for this reverse function.

reverse() is a function.

1st input to reverse is another function (in this case function 'book-detail')

the 2nd input to reverse is the output to function 'book-detail' (as discussed before this function takes the url as input and outputs the actual book, so the output is the id of this book).

str() will make any thing into a string, so for example if id=2 then str(id) will be "2" the string version of 2.

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 = (

('m', 'Maintenance'),

('o', 'On loan'),

('a', 'Available'),

('r', 'Reserved'),

)

status = models.CharField(

max\_length=1,

choices=LOAN\_STATUS,

blank=True,

default='m',

help\_text='Book availability',

)

class Meta:

ordering = ['due\_back']

def \_\_str\_\_(self):

"""String for representing the Model object."""

return f'{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 'm' (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.

Starting with Python 3.6, you can use the string interpolation syntax (also known as f-strings): f'{self.id} ({self.book.title})'.

In older versions of this tutorial, we were using a formatted string syntax, which is also a valid way of formatting strings in Python (e.g. '{0} ({1})'.format(self.id,self.book.title)).

what do they do? so suppose id=2 and title="brave new world" then:

f'{self.id} ({self.book.title})'={0} ({1})'.format(self.id,self.book.title)="2 brave new world"

essentially they format the string by inserting the value of the given variable title and id.

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)

    class Meta:

        ordering = ['last\_name', 'first\_name']

    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 f'{self.last\_name}, {self.first\_name}'

a final model.

imagine a local benefactor donates a number of new books written in another language (say, my own native tongue, 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?

what would be the problem of having this hard-coded? well clearly we can not anticipate every language, for example the local library did not anticipate a book being in Farsi.

what would be the problem of having this as a free text? so in computer security this is terrible. this implies we can literally create a language "gibberish" and have it as the value for the field language. freedom in web development is not a good thing!

that is the problem of having it as a model? it merely complicates our UML, but then again it seems like the best choice for language considering the above.

so for language add the following at the bottom of the models.py file

class Language(models.Model):

"""Model representing a Language (e.g. English, French, Japanese, etc.)"""

name = models.CharField(max\_length=200, help\_text="Enter the book's natural language (e.g. English, French, Japanese etc.)")

def \_\_str\_\_(self):

"""String for representing the Model object (in Admin site etc.)"""

return self.name

also we need to have this as a foreign key in the book class (a book has a foreign key which is the Language).

so we need this line at the end of book class (just before the methods and after the genre field)

language = models.ForeignKey('Language', on\_delete=models.SET\_NULL, null=True)

see below my completed models.py file.

|  |  |
| --- | --- |
|  |  |

from django.db import models

# Create your models here.

class Genre(models.Model):

"""Model representing a book genre."""

name = models.CharField(max\_length=200, help\_text='Enter a book genre (e.g. Science Fiction)')

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.name

class Language(models.Model):

"""Model representing a Language (e.g. English, French, Japanese, etc.)"""

name = models.CharField(max\_length=200, help\_text="Enter the book's natural language (e.g. English, French, Japanese etc.)")

def \_\_str\_\_(self):

"""String for representing the Model object (in Admin site etc.)"""

return self.name

from django.urls import reverse # To generate URLS by reversing 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 has one author, but authors has multiple books

# Author as a string rather than object because it hasn't been declared yet.

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>')

# ManyToManyField used because genre has many books and book has many genres.

# Genre class has already been defined so we can specify the object above.

genre = models.ManyToManyField(Genre, help\_text="Select a genre for this book")

# Language is not declare yet so we use 'Language'

# and it is foreign key which on deletion we delete the record and is not null

language = models.ForeignKey('Language', on\_delete=models.SET\_NULL, null=True)

def get\_absolute\_url(self):

"""Returns the url to access a particular book instance."""

return reverse('book-detail', args=[str(self.id)])

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.title

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='m',

help\_text='Book availability')

class Meta:

ordering = ['due\_back']

def \_\_str\_\_(self):

"""String for representing the Model object."""

return '{0} ({1})'.format(self.id, self.book.title)

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)

class Meta:

ordering = ['last\_name', 'first\_name']

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 '{0}, {1}'.format(self.last\_name, self.first\_name)

Finally recall that after all changes to the model.py we need to re run the migrations. so go save the model.py file. go back to terminal and run the following two lines: (you can copy paste and run both at once, does not really matter)

python3 manage.py makemigrations

python3 manage.py migrate

again for this to work make sure we are in the main project directory (where manage.py is located). so from now on when you see errors such as 'no such file or directory' in the terminal, the first thing to check is that we are in the correct directory (and that the file we are talking about in our arguments does indeed exist.) so in this case we were in the catalog folder, but we need to go back one level, so we had to 'cd ..' before running the above commands.

another common error that can happen to you is that make sure before any interaction takes place with the terminal, we are in the virtual environment, so run 'workon env' before doing the above lines.

it should return an okey status at the end.

# Django Admin site

At this point we'll divert briefly from creating the site, and check out the Django Administration site. It really is totally different to the website (locallibrary) that we are creating, django has created this admin site for us, to handle our locallibrary.

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. So think of this site as the place where the librarian logs in and handles the business. Most real life businesses actually do have such sites that are hidden from average users and secured with passwords so to prevent hacks.

Before we can do any of the tasks above, we need to register our models. To do so either navigate inside the catalog application folder (from terminal access this folder in the main project folder). Then open admin.py file using atom.

Alternatively we can do:

atom ~/django\_project/locallibrary/catalog/admin.py

(please get comfortable with opening files from different folders, from this point onwards I will skip how to open a file and just tell you where the file is and what application to pass it to, for editing purposes we pass the file to atom.)

In this file you should find the below lines of code:

from django.contrib import admin

# Register your models here.

to register our models add these lines at the bottom of admin.py file:

from catalog.models import Author, Genre, Book, BookInstance, Language

admin.site.register(Book)

admin.site.register(Author)

admin.site.register(Genre)

admin.site.register(BookInstance)

admin.site.register(Language)

Note that we have added all the rectangles or models or equally classes that we made in the previous chapter.

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.

Why didn’t we log in the admin site before when we accessed it? Well because we didn’t have a user account first! Any database has data, and for the data to be stored and retrieved there must be a user account with a certain set of privileges defined first. A basic user can only view limited range of data, where as on the other range we can have a super user which can create, modify, view all and delete any data that he or she wants. An admin is a super user. So to log in as the admin we first need to make a super user.

We can do so using manage.py file in the main project folder (main locallibrary folder).

use the below line while you are in the directory containing the manage.py file (again make sure we are using the virtual environment env, or we will get errors about not having django and etc.):

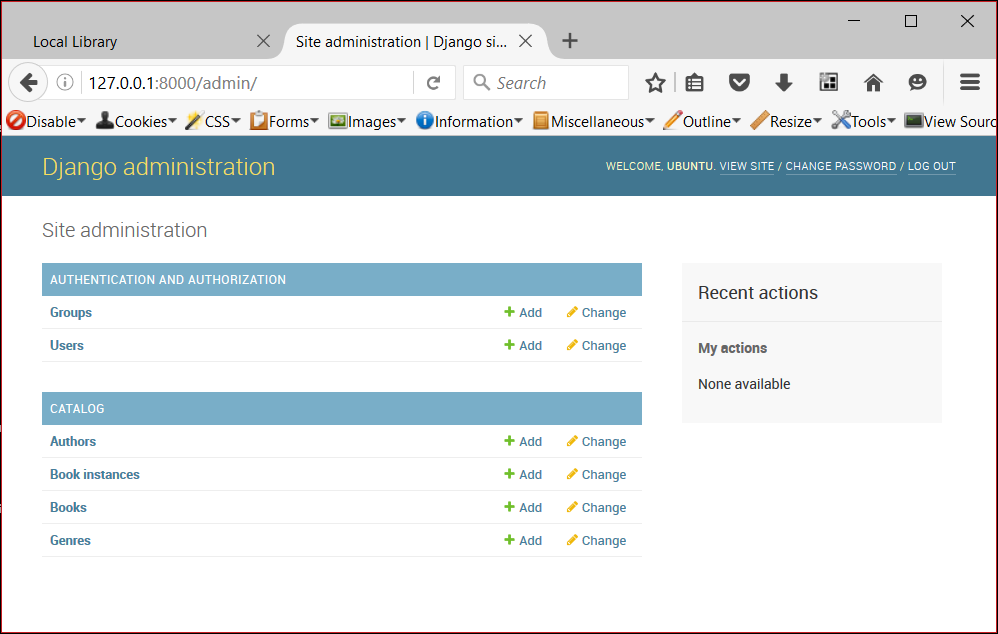
python3 manage.py createsuperuser

You will be prompted to enter a username, email address, and strong password. 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

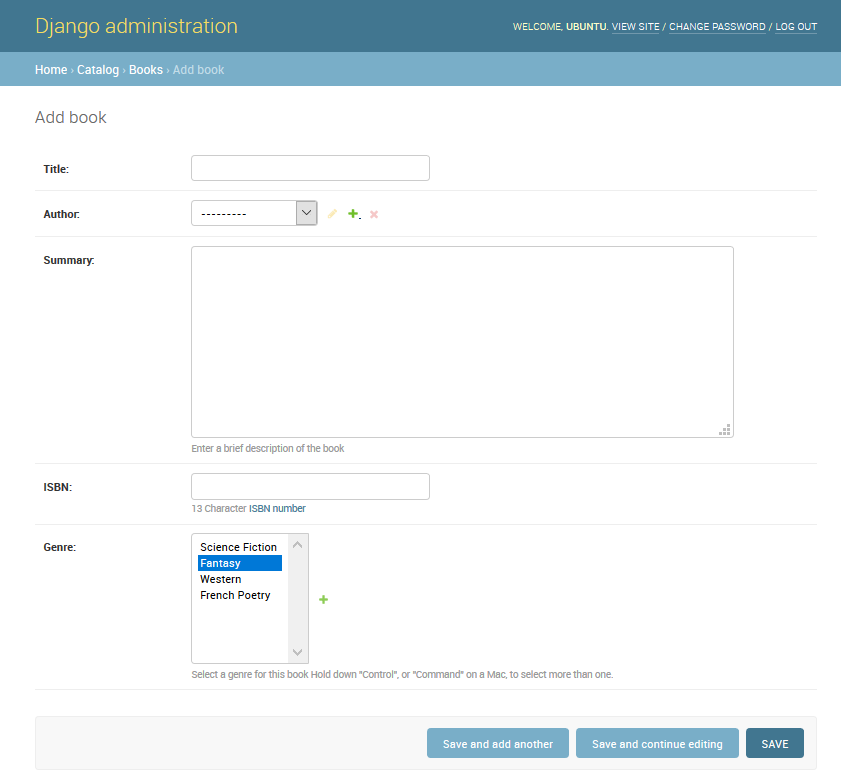
To login to the site, open the /admin URL (e.g. http://127.0.0.1:8000/admin) and enter your new superuser userid and password credentials (you'll be redirected to the login page, and then back to the /admin URL after you've entered your details).

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.



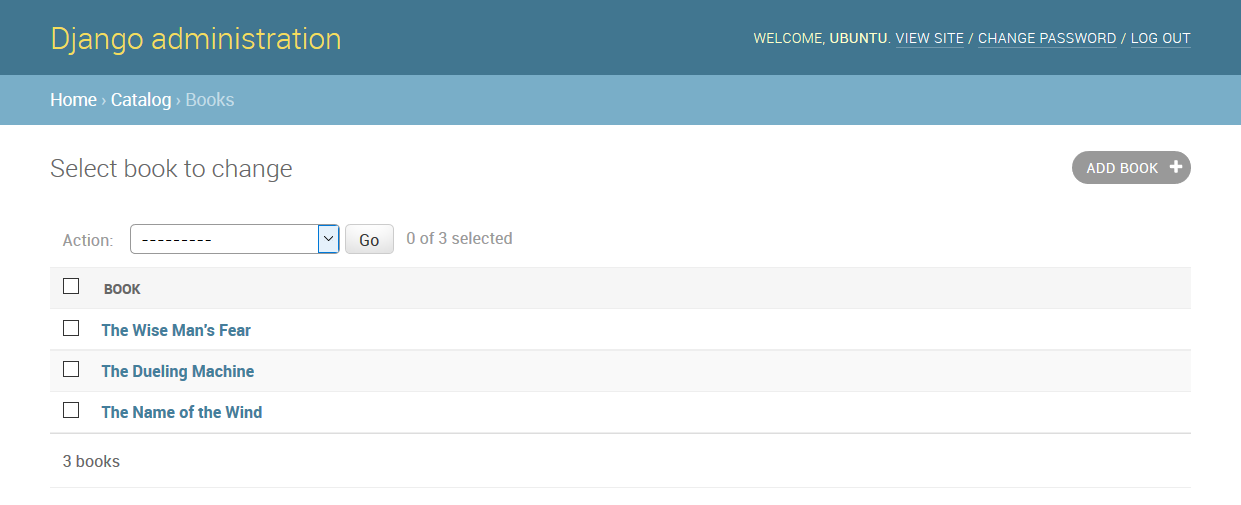
Click on the Add link to the right of Books to create a new book (this will display a dialog much like the one below). Note how the titles of each field, the type of widget used, and the help\_text (if any) match the values you specified in the model.

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.



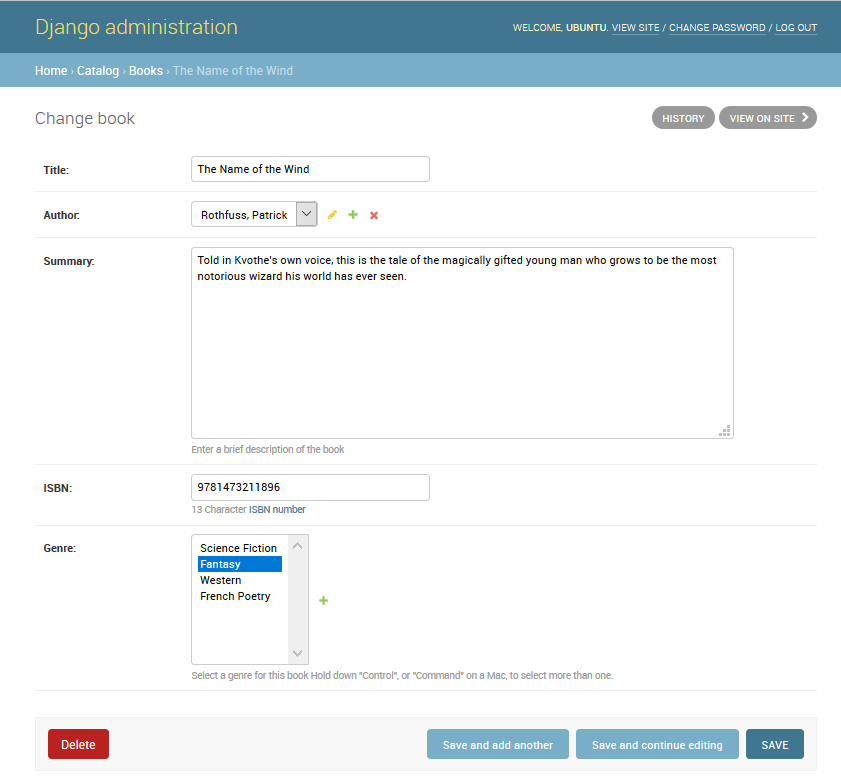
At this point I would like for you to spend some time adding a few books, authors, and genres (e.g. Fantasy) to your application. Make sure that each author and genre includes a couple of different books (this will make your list and detail views more interesting when we implement them later on in the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



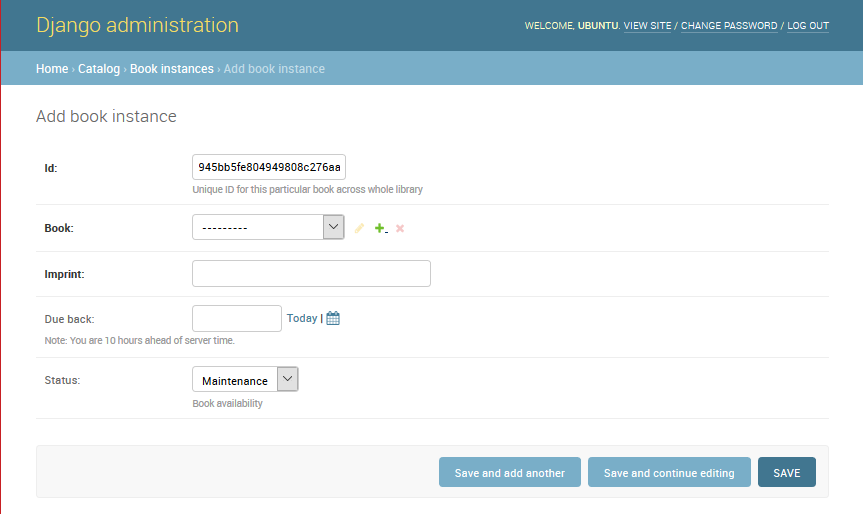
From this list you can delete books by selecting the checkbox next to the book you don't want, selecting the delete... action from the Action drop-down list, and then pressing the Go button. You can also add new books by pressing the ADD BOOK button.

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).



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.



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.

Now let us talk about some advanced configurations of this admin site:

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 at 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!

A complete list of all customisation choices for admin site can be found here: https://docs.djangoproject.com/en/2.1/ref/contrib/admin/

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 (commenting out a line is essentially prefixing it with a # character, as you see in the code below) 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)

Note that pass means an incomplete function that does nothing. We want to define this function (AuthorAdmin) later on.

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 (recall what we said about 'pass') so the admin behaviour will be unchanged! We can now extend these to define our model-specific admin behaviour.

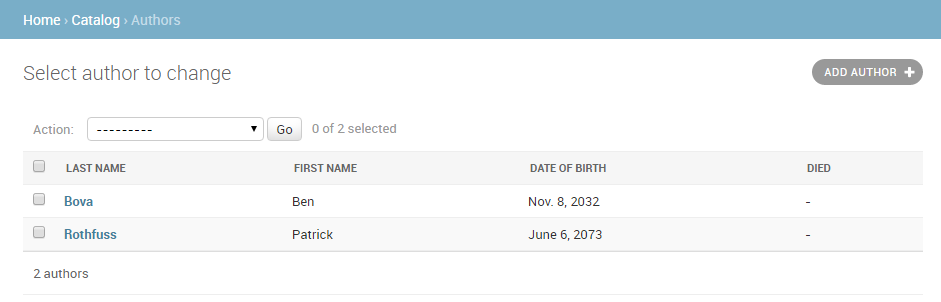
The LocalLibrary currently lists all authors using the object name generated from the model \_\_str\_\_() method. This is fine when you only have a few authors, but once you have many you may end up having duplicates. To differentiate them, or just because you want to show more interesting information 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')

Now navigate to the author list in your website. The fields above should now be displayed, like so: (by this first save the admin file, go back to terminal, go to the man project directoryw here manage.py is, and then do python3 manage.py runserver, then go to browser and open the admin site as we did before. You should find that the authors, book, book instances you recorded before are listed the way the picture below demonstrates. so each autor here has now the first name, dob and dod next to his/her last name)



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, by cost meaning the computer needs to look up the database many times thus slowing the process down, time cost). 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).

Getting the genre may not be a good idea here, because of the "cost" of the database operation. I am 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 in catalog folder, open this again using atom, make sure all changes are saved before closing a file). 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.

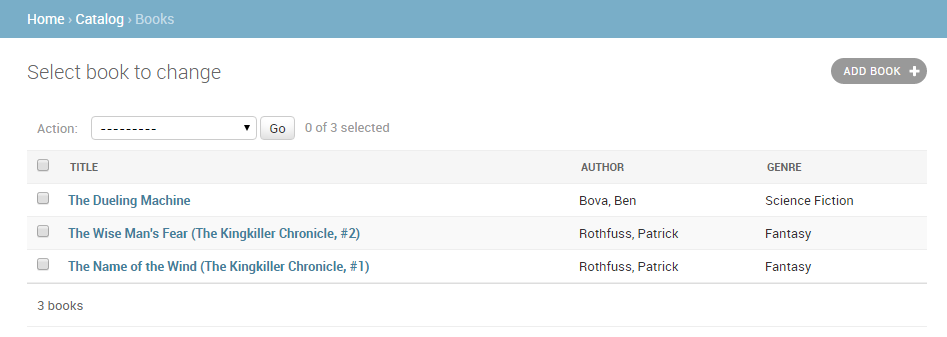
def display\_genre(self):

"""Create 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, open your website and go to the Books list page; you should see a book list like the one below:



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.

Similarly I have updated BookInstance to show at least the status and the expected return date of the book.

See below my admin.py file at this stage:

from django.contrib import admin

# Register your models here.

from catalog.models import Author, Genre, Language, Book, BookInstance

# We have commented out the ones that are overwritten using admin functions.

#admin.site.register(Book)

admin.site.register(Language)

#admin.site.register(BookInstance)

admin.site.register(Genre)

#admin.site.register(Author)

# Define the admin class

class AuthorAdmin(admin.ModelAdmin):

list\_display=('last\_name','first\_name','date\_of\_birth','date\_of\_death')

# Register the admin class with the associated model

admin.site.register(Author, AuthorAdmin)

# Register the Admin classes for Book using the decorator

@admin.register(Book)

class BookAdmin(admin.ModelAdmin):

list\_display=('title','author','display\_genre')

# Register the Admin classes for BookInstace using the decorator

@admin.register(BookInstance)

class BookInstanceAdmin(admin.ModelAdmin):

list\_display=('book','status','due\_back','id')

and find below my models.py at this stage:

from django.db import models

# Create your models here.

from django.urls import reverse # To generate URLS by reversing URL patterns

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

class Language(models.Model):

"""Model representing a Language (e.g. English, French, Japanese, etc.)"""

name = models.CharField(max\_length=200, help\_text="Enter the book's natural language (e.g. English, French, Japanese etc.)")

def \_\_str\_\_(self):

"""String for representing the Model object (in Admin site etc.)"""

return self.name

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 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 a genre can contain many books and a Book can cover many genres.

# Genre class has already been defined so we can specify the object above.

language = models.ForeignKey('Language', on\_delete=models.SET\_NULL, null=True)

def get\_absolute\_url(self):

"""Returns the url to access a particular book instance."""

return reverse('book-detail', args=[str(self.id)])

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.title

def display\_genre(self):

"""Create a string for the Genre. This is required to display genre in Admin."""

return ', '.join(genre.name for genre in self.genre.all()[:3])

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 '{0} ({1})'.format(self.id, self.book.title)

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)

class Meta:

ordering = ['last\_name', 'first\_name']

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):

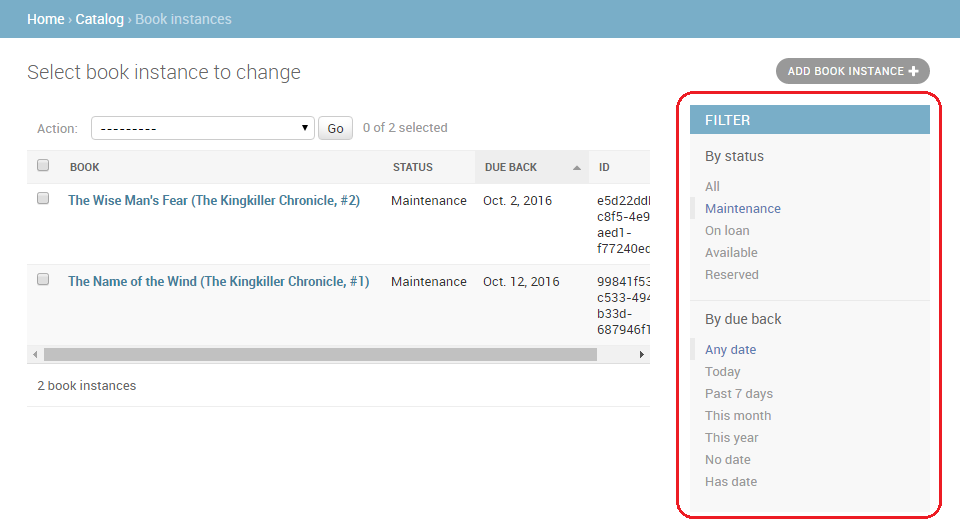
"""String for representing the Model object."""

return '{0}, {1}'.format(self.last\_name, self.first\_name)

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. Add the following line at the top of your BookInstanceAdmin class in admin.py: (so before the list\_display)

**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:



Now lets organise the detail view layout, (where we enter details of a record).

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.

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.

Lets first control which fields are displayed and laid out.

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

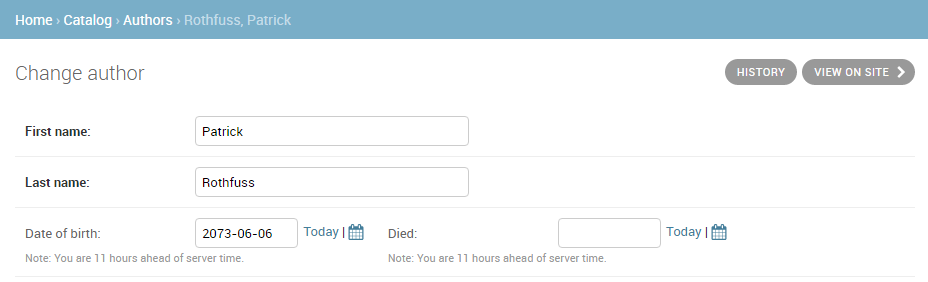
class AuthorAdmin(admin.ModelAdmin):

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).

In your website go to the author detail view — it should now appear as shown below:



Sectioning can be done as well. You can add "sections" to group related model information within the detail form, using the fieldsets attribute as seen below:

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\_display=('book','status','due\_back','id')

list\_filter = ('status', 'due\_back')

**fieldsets = (**

**(None, {**

**'fields': ('book', 'imprint', 'id')**

**}),**

**('Availability', {**

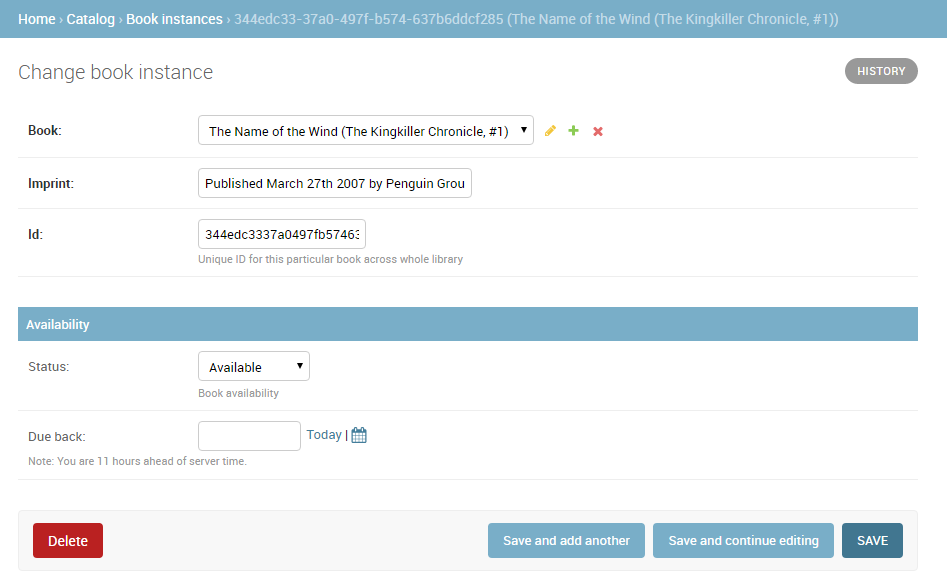
**'fields': ('status', 'due\_back')**

**}),**

**)**

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.

Now navigate to a book instance view in your website; the form should appear as shown below:



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 (horizonal 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**

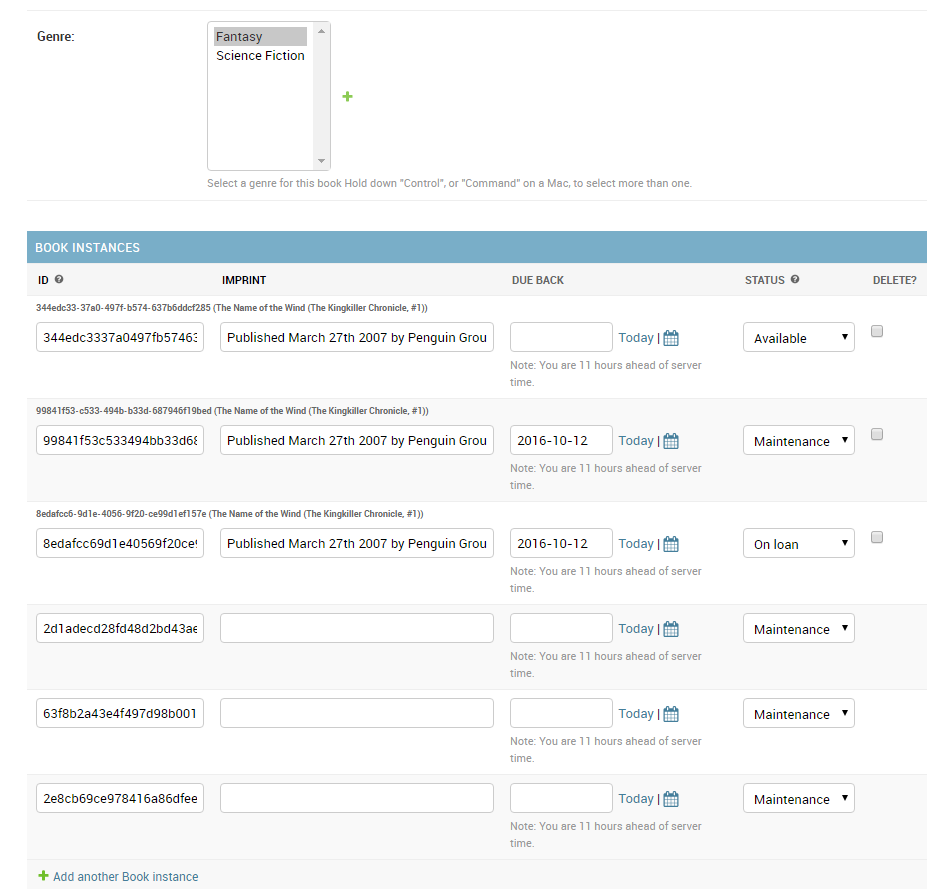
@admin.register(Book)

class BookAdmin(admin.ModelAdmin):

    list\_display = ('title', 'author', 'display\_genre')

**inlines = [BooksInstanceInline]**

Now navigate to a view for a Book in your website — at the bottom you should now see the book instances relating to this book (immediately below the book's genre fields):



n 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. For more information see: https://docs.djangoproject.com/en/dev/ref/contrib/admin/#django.contrib.admin.TabularInline

I have copied my admin.py at this stage, compare yours to mines, should look similar, if you got any errors from python3 manage.py runserver, please check spelling and spacing of the code you wrote.

from django.contrib import admin

# Register your models here.

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

# We have commented out the ones that are overwritten using admin functions.

#admin.site.register(Book)

admin.site.register(Language)

#admin.site.register(BookInstance)

admin.site.register(Genre)

#admin.site.register(Author)

# Define the admin class

class BooksInline(admin.TabularInline):

"""Defines format of inline book insertion (used in AuthorAdmin) """

model=Book

class AuthorAdmin(admin.ModelAdmin):

"""Administration object for Author models.

Defines:

- fields to be displayed in list view (list\_display)

- orders fields in detail view (fields), grouping the date fields horizontally

- adds inline addition of book in author view (inlines) [scroll to the bottom

of an author page to add books written by him/her]

"""

list\_display=('last\_name','first\_name','date\_of\_birth','date\_of\_death')

fields=['first\_name','last\_name', ('date\_of\_birth', 'date\_of\_death')]

inlines=[BooksInline]

# Register the admin author class with the associated model

admin.site.register(Author, AuthorAdmin)

class BookInstanceInline(admin.TabularInline):

"""Defines format of inline book instance insertion used in (BookAdmin)"""

model = BookInstance

class BookAdmin(admin.ModelAdmin):

"""Administration object for Book models:

Defines:

- fields to be displayed in list view (list\_display)

- adds inline addition of book instances in book view (inlines)

"""

list\_display=('title','author','display\_genre')

inlines = [BookInstanceInline]

# Register the admin book class with associated model

admin.site.register(Book, BookAdmin)

class BookInstanceAdmin(admin.ModelAdmin):

"""Administration object for BookInstace models.

Defines:

- fields to be displayed in list view (list\_display)

- filters that will be displayed in sidebar (list\_filter)

- grouping of fields into sections (fieldsets)

"""

list\_display=('book','status','due\_back','id')

list\_filter=('status', 'due\_back')

fieldsets=(

(None, {

'fields': ('book', 'imprint', 'id')

}),

('Availability',{

'fields': ('status', 'due\_back')

}),

)

# Register the Admin classes for BookInstace

admin.site.register(BookInstance)

for reference this is my models.py file at this stage, yours should look more or less same as this right now:

from django.db import models

# Create your models here.

from django.urls import reverse # To generate URLS by reversing URL patterns

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

class Language(models.Model):

"""Model representing a Language (e.g. English, French, Japanese, etc.)"""

name = models.CharField(max\_length=200, help\_text="Enter the book's natural language (e.g. English, French, Japanese etc.)")

def \_\_str\_\_(self):

"""String for representing the Model object (in Admin site etc.)"""

return self.name

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 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 a genre can contain many books and a Book can cover many genres.

# Genre class has already been defined so we can specify the object above.

language = models.ForeignKey('Language', on\_delete=models.SET\_NULL, null=True)

def get\_absolute\_url(self):

"""Returns the url to access a particular book instance."""

return reverse('book-detail', args=[str(self.id)])

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.title

def display\_genre(self):

"""Create a string for the Genre. This is required to display genre in Admin."""

return ', '.join(genre.name for genre in self.genre.all()[:3])

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 '{0} ({1})'.format(self.id, self.book.title)

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)

class Meta:

ordering = ['last\_name', 'first\_name']

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 '{0}, {1}'.format(self.last\_name, self.first\_name)

That is pretty much all the changes realistically we make to the admin site, beyond that, the web would be a good source to find more modifications.

Note the entire point of this section was to make the job of the librarian easier, he or she is not a tech savvy person, so the interface should be fool proof, logically safe, and easy to use.

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 BooksInstanceInline model. You can try it yourself, but this is questions that a developer should be asking him or herself.

# Home Page

We're now ready to add the code that displays our first complete page — a home page for the LocalLibrary website. The home page will show the number of records we have for each model type and provide 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.

After we defined our models and created some initial library records to work with, it's time to write the code that presents that information to users. The first thing we need to do is determine what information we want to display in our pages, and define the URLs to use for returning those resources. Then we'll create a URL mapper, views, and templates to display the pages.

The following diagram describes the main data flow, and the components required when handling HTTP requests and responses. As we already implemented the model, the main components we'll create are:

* URL mappers 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 HTML pages that display the data, and return the pages to the user to view in the browser.
* Templates to use when rendering data in the views.



The http request comes from the user (when clicking on a link in the browser or opening a url, the browser will send this request on behalf of the user), which is then through internet travels to the server, or django in this case, django searches through urls to find a matching url, as discussed in the URL intro section, this then calls a function to generate the correct view from views.py. views.py might in turn look up some data (or write some data in some cases) from the data base (by talking to models.py). views might also load an html from templates, and populate it with data to be passed as the response to the request from the user. The response similarly travels back through internet to the user's browser and gets displayed. This is pretty much how internet browsing works.

As you'll see in the next section, I will have 5 pages to display, which is too much information to document in a single article. Therefore, this section, I will focus on how to implement the home page, and I'll cover 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.

As this version of LocalLibrary is essentially read-only for end users (the users do not modify the data in our databsae, they only view it for example a user cannot view the website to change the summary of a book on the website), 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'll need for our pages are:

catalog/ — The home (index) page.

catalog/books/ — A list of all books.

catalog/authors/ — A list of all authors.

catalog/book/<id> — The detail view for a particular book, with a field primary key of <id> (the default). For example, the URL for the third book added to the list will be /catalog/book/3.

catalog/author/<id> — The detail view for the specific author with a primary key field of <id>. For example, the URL for the 11th author added to the list will be /catalog/author/11.

The first three URLs will return the index page, books list, and authors list. These URLs do not encode any additional information, and the queries that fetch data from the database will always be the same. However, the results that the queries return will depend on the contents of the database.

By contrast the final two URLs will display detailed information about a specific book or author. These URLs encode the identity of the item to display (represented by <id> above). The URL mapper will extract the encoded information and pass it to the view, and the view will dynamically determine what information to get from the database. By encoding the information in the URL we will use a single set of a url mapping, a view, and a template to handle all books (or authors).

With Django, you can construct your URLs however you require — you can encode information in the body of the URL as shown above, or include GET parameters in the URL, for example /book/?id=6. What is get? recall how we made forms in html, on top of the form we have a method attribute, method of submitting a form can either be GET: https://www.w3schools.com/html/tryit.asp?filename=tryhtml\_form\_get

or POST:

https://www.w3schools.com/html/tryit.asp?filename=tryhtml\_form\_post

The main difference is visibility and security between the two. In general I have learnt to use GET only to learn how things work but POST in real life applications.

Read this article for a full comparison: https://www.diffen.com/difference/GET-vs-POST-HTTP-Requests

As you can see in the article above, GET requires for us to encode data in the URL itself thus we need to prepare views.py for this. That is why we have things like ?id=5 in the url. Look at this for a little bit more info: https://stackoverflow.com/questions/5998425/url-format-with-get-parameters

Whichever approach you use, the URLs should be kept clean, logical, and readable, and more importantly should hide anything that is not necessary. More info you show the higher the chance a hacker can hack the website.

The Django documentation recommends encoding information in the body of the URL to achieve better URL design (i.e. using GET!).

Now let us describe making of the index page.

The first page we'll create is the index page (catalog/). The index page will include some static HTML, along with generated "counts" of different records in the database. To make this work we'll create a URL mapping, a view, and a template.

Please make sure you read this section at least twice, it is very crucial.

When we created the skeleton website, we updated the locallibrary/urls.py file to ensure that whenever an URL that starts with catalog/ is received, the URLConf module catalog.urls will process the remaining substring. (recall we added this during URL intro section of this tutorial)

Try to open this file, it is located at (~/django\_project/locallibrary/locallibrary/urls.py) open it with atom.

You can find the following code snippet from locallibrary/urls.py includes the catalog.urls module:

urlpatterns += [

path('catalog/', include('catalog.urls')),

]

This essentially diverts all requests with url like websitename/catalog to the urls.py file inside catalog folder.

We also created a placeholder file for the URLConf module, named /catalog/urls.py. Add the following bold lines to that file: (recall this file was created before in the URLs intro section, just add the highlighted lines at this moment.)

urlpatterns = [

**path('', views.index, name='index'),**

]

The path() function defines the following:

1) A URL pattern, which is an empty string: ''. We'll discuss URL patterns in detail when working on the other views.

2) A view function that will be called if the URL pattern is detected: views.index, which is the function named index() in the views.py file.

For now remember that in urls.py file we need each path to have this two essential components, a function to call in views.py and a url to be mapped to that function.

The path() function also specifies a name parameter, which is a unique identifier for this particular URL mapping. You can use the name to "reverse" the mapper, i.e. to dynamically create a URL that points to the resource that the mapper is designed to handle. For example, we can use the name parameter to link to our home page from any other page by adding the following link in a template:

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

remember that the name 'index' is unique in all entire website. It is an identifier for this url to function mapping. So it can simply be used within the href of a link to direct the user's browser to the output of the mapped function.

We can hard code the link as in <a href="/catalog/">Home</a>), but if we change the pattern for our home page, for example, to /catalog/index) the templates will no longer link correctly. Using a reversed url mapping is much more flexible and robust. (it is always a good practice in programming to reference the variable name as opposed to the value of the variable. If we reference to variable in many different places in our project, by changing the variable's value once all references are automatically updated which is much easier than changing the values wherever this was referenced. )

A view is a function that processes an HTTP request, fetches the required data from the database, renders the data in an HTML page using an HTML template, and then returns the generated HTML in an HTTP response to display the page to the user. The index view follows this model — it fetches information about the number of Book, BookInstance, available BookInstance and Author records that we have in the database, and passes that information to a template for display.

Open catalog/views.py using atom, and note that the file already imports the render() shortcut function to generates HTML file using a template and data:

from django.shortcuts import render

# Create your views here.

Paste the following lines at the bottom of the file:

from catalog.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()

# The 'all()' is implied by default.

num\_authors = Author.objects.count()

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

The first line imports the model classes that we'll use to access data in all our views.

The first part of the view function fetches the number of records using the objects.all() attribute on the model classes. It also gets a list of BookInstance objects that have a value of 'a' (Available) in the status field. We talked more about how to access model data in the previous section (recall we had things like:

Book.object.filter(title\_\_contains='wild')

Book.object.filter(genre\_\_name\_\_icontains='fiction')

Book.object.filter(type\_\_cover\_\_name\_\_exact='hard')

... where the double underscore (\_ \_) is used to navigate between the relationships as seen above.

)

Next we have created a context varibale holding a dictionary of data that we have retrieved before. Please check this link to learn about dictionaries and get comfortable with notations such as 'key' and 'value'.

Here in the context dictionary the keys are 'num\_books', 'num\_instances', ... and the values of these keys are the actual number of books in our database, the number of book instances in total (for all books), and so on.

Why did we do this? because this is the data that needs to be presented on the index page, we are aiming to show the user in the main page how many books, authors and so on we have.

So at the end of the view function we call the render() function to create an HTML page and return the page as a response. This shortcut function wraps a number of other functions to simplify a very common use case. The render() function accepts the following parameters:

* the original request object, which is an HttpRequest.
* an HTML template with placeholders for the data.
* A context variable, which is a Python dictionary, containing the data toinsert into the placeholders.

I'll talk more about templates and the context variable in the next section. Let's get to creating our template so we can actually display something to the user!

A template is a text file that defines the structure or layout of a file (such as an HTML page), using placeholders to represent actual content. Django will automatically look for templates in a directory named 'templates' in your application. For example, in the index view that we just added, the render() function will expect to find the file index.html in /locallibrary/catalog/templates/ and will raise an error if the file is not present. You can check by saving the previous changes and accessing 127.0.0.1:8000 in your browser - it will display a fairly intuitive error message: "TemplateDoesNotExist at /catalog/", and other details.

The index template will need standard HTML markup for the head and body, along with navigation sections to link to the other pages of the site that we haven't created yet, and to sections that display introductory text and book data. Much of the HTML and navigation structure will be the same in every page of our site. Instead of duplicating boilerplate code on every page, you can use the Django templating language to declare a base template, and then extend it to replace just the bits that are different for each specific page. (this concept is also called inheritance, for example a mammal gives birth and breathes and so on, a canine has fur and a tail but is also implemented from a mamma as it is a mammal essentially, a dog is implemented from a canine, so a dog is in turn implemented from a mammal and finally you can say a Labrador is implemented from a dog but it extends it to a certain specific case. In this case we make a base template, then all other templates will be implemented by that, or in turn they extend the base template in some way, again we might have more templates that further extend the previous level)

The following code snippet is a sample base template from a base\_generic.html file. The sample includes common HTML with sections for a title, a sidebar, and main contents marked with the named block and endblock template tags, shown in bold. You can leave the blocks empty, or include default content to use when rendering pages derived from the template.

<!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>

The base template is then extended, by extension we will actually add material where currently the {% block %} text is. Please note how we start a block with a name and then we end it with {% endblock %}. The code between the opening and ending of a block, is in <!-- --> which is a comment, note that this comment is different to the way we made comments in python (Comments in html are inside <!--comment-->, comments in python are inside """comment""")

When defining a template for a particular view, we first specify the base template using the extends template tag — see the code sample below. Then we declare what sections from the template we want to replace (if any), using block/endblock sections as in the base template.

For example, the code snippet below shows how to use the extends template tag and override the content block. The generated HTML will include the code and structure defined in the base template, including the default content you defined in the title block, but the new content block in place of the default one.

{% extends "base\_generic.html" %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary, a website developed by <em>Mozilla Developer Network</em>!</p>

{% endblock %}

We will use the following code snippet as the base template for the LocalLibrary website. As you can see, it contains some HTML code and defines blocks for title, sidebar, and content. We have a default title and a default sidebar with links to lists of all books and authors, both enclosed in blocks to be easily changed in the future.

I also introduced two additional template tags: url and load static. These tags will be explained in following sections. For now just think of static as the keyword to mention before including anything such as javascript, css, or image or audio. They are static as they do not change no matter what data we have on the database, also that they lie on the directory statically. url is used when referencing a variable (recall the section we talked about paths that have special unique names that are declared once).

Create a new file "base\_generic.html" in /locallibrary/catalog/templates/ and paste the following code to the file: (to do so I recommend that you cd inside this directory, then using command "atom base\_generic.html" create this file, paste the below content and save it.)

<!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://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<!-- 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 includes 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 displays well on different screen sizes. To further know about how bootstrap works I suggest this link to view: https://www.w3schools.com/bootstrap4/

The base template also references a local css file (styles.css) that provides additional styling. Create a styles.css file in /locallibrary/catalog/static/css/ and paste the following code in the file: (again do this by making directories in terminal by first going to catalog directory using cd .., then mkdir static then cd inside static, then mkdir css, then cd inside css and finally use atom styles.css to create this file.)

.sidebar-nav {

margin-top: 20px;

padding: 0;

list-style: none;

}

Create a new HTML file index.html in /locallibrary/catalog/templates/ and paste the following code in the file. This code extends our base template on the first line, and then replaces the default content block for the template.

{% extends "base\_generic.html" %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary</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 %}

n the Dynamic content section we declare placeholders (template variables) for the information from the view that we want to include. The variables are enclosed with double brace (handlebars {{}} ), as shown in bold in the code sample.

You can easily recognise template variables and template tags (functions) - variables are enclosed in double braces ({{ num\_books }}), and tags are enclosed in single braces with percentage signs ({% extends "base\_generic.html" %}).

The important thing to note here is that variables are named with the keys that we pass into the context dictionary in the render() function of our view (see sample below actually taken from our views.py file). Variables will be replaced with their associated values when the template is rendered.

context = {

'**num\_books**': num\_books,

'**num\_instances**': num\_instances,

'**num\_instances\_available**': num\_instances\_available,

'**num\_authors**': num\_authors,

}

return render(request, 'index.html', context=context)

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 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 the template library, as shown in the code sample below (this is taken from what you included in the main generic template before, do not add it again!). You can then use the static template tag and specify the relative URL to the required file.

<!-- Add additional CSS in static file -->

{% load static %}

<link rel="stylesheet" href="{% static 'css/styles.css' %}">

You can add an image into the page in a similar way, for example:

(again just as an example)

{% load static %}

<img src="{% static 'catalog/images/local\_library\_model\_uml.png' %}" alt="UML diagram" style="width:555px;height:540px;">

For more information on working with static files see: https://docs.djangoproject.com/en/2.1/howto/static-files/

The base template above introduced the url template tag. Let us expand on this: (this snippet is taken from the generic template)

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

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

You need to point Django to search for your templates in the templates folder. To do that, add the templates dir to the TEMPLATES object by editing the settings.py file as shown in bold in the following code sample:

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [

**os.path.join(BASE\_DIR, 'templates'),**

        ],

        'APP\_DIRS': True,

        'OPTIONS': {

            'context\_processors': [

                'django.template.context\_processors.debug',

                'django.template.context\_processors.request',

                'django.contrib.auth.context\_processors.auth',

                'django.contrib.messages.context\_processors.messages',

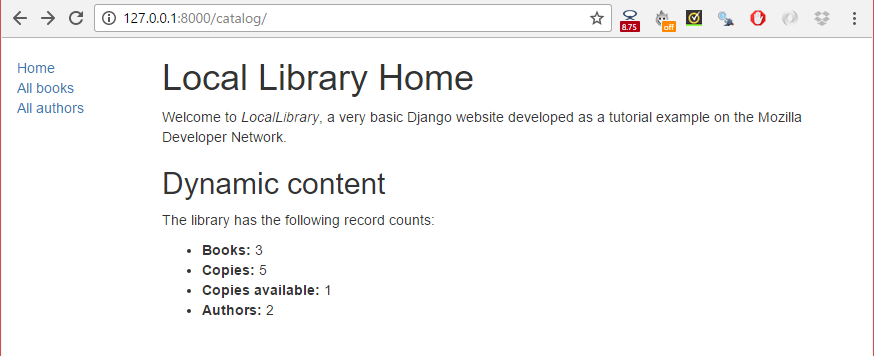
            ],

        },

    },

]

Okey what does our website look like now? At this point we have created all required resources to display the index page. Run the server (python3 manage.py runserver) and open http://127.0.0.1:8000/ in your browser. If everything is configured correctly, your site should look like the following screenshot. (again recall we need to be in env, be in correct directory and have everything saved up beforehand)



Note that the number of books, copies and etc might be different in yours, as it depends what books you have done in the admin site in the previous section. (maybe try going to admin site, add more books, then revisiting this home page to observe the changes.)

The All books and All authors links will not work yet because the paths, views, and templates for those pages are not defined. We just inserted placeholders for those links in the base\_generic.html template.

I have also done some extra work. I changed the title of the index page by extending the title block of the base template. I have also added some functions in view to generate counts for genres and book that contain a particular word (case sensitive). They then pass the result to context which is injected to index.html and rendered.

Find below my index.html: (located at ~/django\_project/locallibrary/catalog/templates)

{% extends "base\_generic.html" %}

{% block title %}<title>Local Library Home Page</title>{% endblock %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary</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>

<li><strong>Books that have blah in their title:</strong> {{ num\_blah\_books }}</li>

<li><strong>Genres starting with sceince:</strong> {{ num\_sceince\_genres }}</li>

</ul>

{% endblock %}

this is my views.py: (located at ~/django\_project/locallibrary/catalog/)

from django.shortcuts import render

# Create your views here.

from catalog.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()

# The 'all()' is implied by default.

num\_authors = Author.objects.count()

# Count all the books that have 'blah' in their titles

num\_blah\_books = Book.objects.filter(title\_\_contains='blah').count()

# Count all genres that have 'science' at the beginning

num\_sceince\_genres = Genre.objects.filter(name\_\_startswith='science').count()

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

'num\_blah\_books': num\_blah\_books,

'num\_sceince\_genres': num\_sceince\_genres,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

this is my models.py page (I have not changed it from the last chapter, this is only for reference, and is located in ~/django\_project/locallibrary/catalog/)

from django.db import models

# Create your models here.

from django.urls import reverse # To generate URLS by reversing URL patterns

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

class Language(models.Model):

"""Model representing a Language (e.g. English, French, Japanese, etc.)"""

name = models.CharField(max\_length=200, help\_text="Enter the book's natural language (e.g. English, French, Japanese etc.)")

def \_\_str\_\_(self):

"""String for representing the Model object (in Admin site etc.)"""

return self.name

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 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 a genre can contain many books and a Book can cover many genres.

# Genre class has already been defined so we can specify the object above.

language = models.ForeignKey('Language', on\_delete=models.SET\_NULL, null=True)

def get\_absolute\_url(self):

"""Returns the url to access a particular book instance."""

return reverse('book-detail', args=[str(self.id)])

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.title

def display\_genre(self):

"""Create a string for the Genre. This is required to display genre in Admin."""

return ', '.join(genre.name for genre in self.genre.all()[:3])

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 '{0} ({1})'.format(self.id, self.book.title)

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)

class Meta:

ordering = ['last\_name', 'first\_name']

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 '{0}, {1}'.format(self.last\_name, self.first\_name)

And this is my settings.py (located at ~/django\_project/locallibrary/ locallibrary/)

"""

Django settings for locallibrary project.

Generated by 'django-admin startproject' using Django 2.1.4.

For more information on this file, see

https://docs.djangoproject.com/en/2.1/topics/settings/

For the full list of settings and their values, see

https://docs.djangoproject.com/en/2.1/ref/settings/

"""

import os

# Build paths inside the project like this: os.path.join(BASE\_DIR, ...)

BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = '8wm5sw$-+kv57b2f)blyai)$tiya\*$5p00ztt#0t0g#((qx@l@'

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'catalog.apps.CatalogConfig'

]

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'locallibrary.urls'

# Note that we have added the directorys of templates to be locallibrary/catalog/templates

# Django needs to be directed to specifically to find these.

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [

os.path.join(BASE\_DIR, 'templates'),

],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

WSGI\_APPLICATION = 'locallibrary.wsgi.application'

# Database

# https://docs.djangoproject.com/en/2.1/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

# Password validation

# https://docs.djangoproject.com/en/2.1/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

{

'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

},

]

# Internationalization

# https://docs.djangoproject.com/en/2.1/topics/i18n/

LANGUAGE\_CODE = 'en-us'

TIME\_ZONE = 'Europe/London'

USE\_I18N = True

USE\_L10N = True

USE\_TZ = True

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/2.1/howto/static-files/

STATIC\_URL = '/static/'

Note the very last line of this file, clearly states that all static files such as css, images, and everything else must be under the static directory inside catalog directory. (css can be itself under another directory but it must be under static in catalog)

My files currently look as below:

~/django\_project/

locallibrary/

catalog/

migrations/

\_\_init\_\_.py

...

\_\_pycache\_\_/

...

static/

style.css

template/

base\_generic.html

index.html

admin.py

apps.py

\_\_init\_\_.py

models.py

tests.py

urls.py

views.py

locallibrary/

\_\_pycache\_\_/

...

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

db.sqlite3

manage.py

the / denotes the file is actually a directory, the content of the directory are indented in the following line. for example catalog/ is a directory and static/ is a directory inside catalog which itself contains only style.css at this stage.

We just created the home page for our site — an HTML page that displays a number of records from the database and links to other yet-to-be-created pages. Along the way we learned fundamental information about url mappers, views, querying the database with models, passing information to a template from a view, and creating and extending templates.

In the next article we'll build upon this knowledge to create the remaining four pages of our website.

# Generic list and detail views

In this section we are aiming to create a page for books and similarly authors. I will explain you through the books page, but for authors page I will only show you the final completed code as it is very similar.

The process is similar to 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 (think of amazon or argos website, they have millions of lists, for example kitchen appliances under 20 pounds, over 40, tvs under 30 inches, under 40, they are all separate lists ...argos or amazon do not create individual views for every single one of these, rather they use patterns in URLs to modify a generic view properly). 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 section will demonstrate how to paginate (split data into pages) your data when using generic class-based list views.

Open /catalog/urls.py and copy in the line shown in bold below. As for the index page, this path() function defines a pattern to match against the URL ('books/'), a view function that will be called if the URL matches (views.BookListView.as\_view()), and a name for this particular mapping. Recall we added the existing line in this file before when we wanted to map 'websitename/catlog/' to index function in views.py.

urlpatterns = [

path('', views.index, name='index'),

**path('books/', views.BookListView.as\_view(), name='books'),**

]

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.

Please first get comfortable with python classes at this stage using this tutorial: https://www.w3schools.com/python/python\_classes.asp

Still confused? Look at index(request) function inside of views.py (located at catalog/views.py), it is clear to see in our urls.py we had a path that calls up this function and passes the request automatically to it. Similarly our new path (the one you added in this section) must be calling a relevant function, thus we use the as\_view() functionality to create a class instance first, then use the function within it.

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 that 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

You do not need to add the above lines, however it is up to you to experiment, though beware you might need to change things later on in this tutorial to match the above declarations. For example if you change the template\_name we should refer to it by the new name and so on.

Before we move on, please read the following tutorial on class inheritence and overriding methods in python, it will make things clearer:

https://www.programiz.com/python-programming/inheritance

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 the context

        context = super(BookListView, self).get\_context\_data(\*\*kwargs)

        # Create any data and add it to the context

        context['some\_data'] = 'This is just some data'

        return context

To learn more about what the super function is and how a sub class can communicate with the super class it was built on look at:

<https://www.pythonforbeginners.com/super/working-python-super-function>

And to see what \*\*kwargs is look at:

<https://pythontips.com/2013/08/04/args-and-kwargs-in-python-explained/>

They are not too necessary to learn but it is good for you to understand how the code above actually works. Context variable is already hard fixed from the super class generic. Our only way of modifying it is to actually get this from the super and do any changes within the body of our subclass.

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.

For more examples of what you can do with the built-in generic views check out this tutorial:

<https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/>

Okey let us move on to the template that will be using this view class. Create the appropriate directory using mkdir, we want now to have(essentially add the folder in bold):

~/django\_project/locallibrary/catalog**/**templates**/catalog/**

Create the HTML file inside the new folder you just added (so: /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).

{% 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.

We use the if, else, and endif template tags to check whether the book\_list has been defined and is not empty. If book\_list is empty, then the else clause displays text explaining that there are no books to list. If book\_list 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:

<https://docs.djangoproject.com/en/2.1/ref/templates/builtins/>

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.

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 from an html file!)

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!

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>

You won't be able to build the 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.

Let us now make the book detail page. 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 also to confirm whether/when it is available.

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

urlpatterns = [

path('', views.index, name='index'),

    path('books/', views.BookListView.as\_view(), name='books'),

  path('book/<int:pk>', views.BookDetailView.as\_view(), name='book-detail'),

]

For the book-detail path the URL pattern uses a special syntax to capture the specific id of the book that we want to see. The syntax is very simple: angle brackets define the part of the URL to be captured, enclosing the name of the variable that the view can use to access the captured data. For example, <something> , will capture the marked pattern and pass the value to the view as a variable "something". You can optionally precede the variable name with a converter specification that defines the type of data (int, str, slug, uuid, path).

In this case we use '<int:pk>' to capture the book id, which must be a specially formatted string and pass it to the view as a parameter named pk (short for primary key). This is the id that is being used to store the book uniquely in the database, as defined in the Book Model.

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

The generic class-based detail view expects to be passed a parameter named pk. If you're writing your own function view you can use whatever parameter name you like, or indeed pass the information in an unnamed argument. (using \*args or \*\*kwargs)

We still need to have an actual view for the book details in our view.py file. 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. (recall how we descibed doing this before in this section)

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, primary\_key):

try:

book = Book.objects.get(pk=primary\_key)

except Book.DoesNotExist:

raise Http404('Book does not exist')

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

To learn more about what that try and except is look at this link:

<https://www.pythonforbeginners.com/error-handling/python-try-and-except>

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).

Alternatively, we can use the get\_object\_or\_404() function as a shortcut to to raise an Http404 exception if the record is not found.

from django.shortcuts import get\_object\_or\_404

def book\_detail\_view(request, primary\_key):

  book = get\_object\_or\_404(Book, pk=primary\_key)

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

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).

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") models, 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()).

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.

Beware also that if you don't define an order (on your class-based view or model), you will also see errors from the development server like this one:

[29/May/2017 18:37:53] "GET /catalog/books/?page=1 HTTP/1.1" 200 1637

/foo/local\_library/venv/lib/python3.5/site-packages/django/views/generic/list.py:99: UnorderedObjectListWarning: Pagination may yield inconsistent results with an unordered object\_list: <QuerySet [<Author: Ortiz, David>, <Author: H. McRaven, William>, <Author: Leigh, Melinda>]>

allow\_empty\_first\_page=allow\_empty\_first\_page, \*\*kwargs)

That happens because the paginator object expects to see some ORDER BY being executed on your underlying database. Without it, it can't be sure the records being returned are actually in the right order!

This tutorial didn't reach Pagination (yet, but soon enough), but since you can't use sort\_by() and pass a parameter (the same with filter() described above) you will have to choose between three choices:

1)Add a ordering inside a class Meta declaration on your model.

2)Add a queryset attribute in your custom class-based view, specifying a order\_by().

3)Adding a get\_queryset method to your custom class-based view and also specify the order\_by().

If you decide to go with a class Meta for the Author model (probably not as flexible as customizing the class-based view, but easy enough), you will end up with something like this:

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 f'{self.last\_name}, {self.first\_name}'

class Meta:

ordering = ['last\_name']

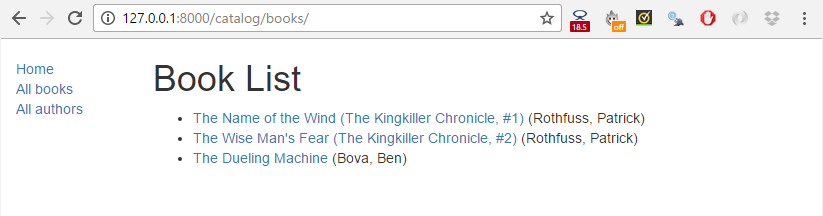
Of course, the field doesn't need to be last\_name: it could be any other.

And last, but not least, you should sort by an attribute/column that actually has a index (unique or not) on your database to avoid performance issues. Of course, this will not be necessary here (and we are probably getting ourselves too much ahead) with so few books (and users!), but it is something to keep in mind for future projects.

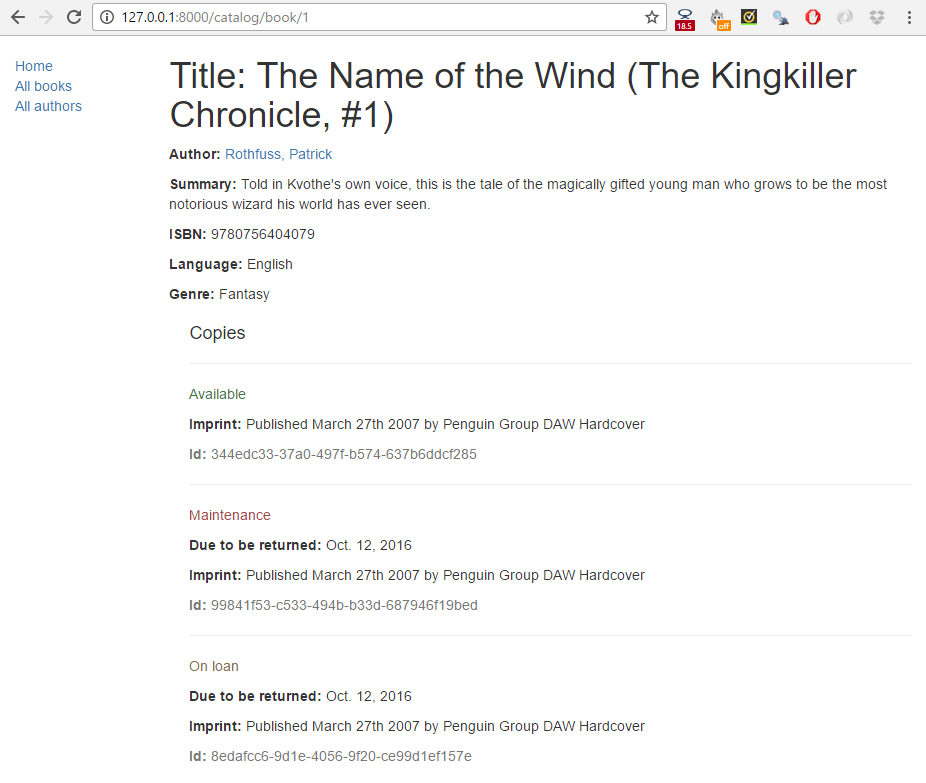
At this point, we should have created everything needed to display both the book list and book detail pages. Run the server (python3 manage.py runserver) and open your browser to <http://127.0.0.1:8000/>.

Don't click any author or author detail links yet!

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.



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 inbuilt 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!

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.

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">

<p>Page {{ page\_obj.number }} of {{ page\_obj.paginator.num\_pages }}.</p>

</span>

{% if page\_obj.has\_next %}

<a href="{{ request.path }}?page={{ page\_obj.next\_page\_number }}">next</a>

{% endif %}

</span>

</div>

{% endif %}

{% endblock %}

My base\_generic.html currently looks like:

<!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://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<!-- 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="{% url 'books' %}">All books</a></li>

<li><a href="">All authors</a></li>

</ul>

{% endblock %}

</div>

<div class="col-sm-10 ">{% 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">

<p>Page {{ page\_obj.number }} of {{ page\_obj.paginator.num\_pages }}.</p>

</span>

{% if page\_obj.has\_next %}

<a href="{{ request.path }}?page={{ page\_obj.next\_page\_number }}">next</a>

{% endif %}

</span>

</div>

{% endif %}

{% endblock %}

</div>

</div>

</div>

</body>

</html>

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.

That's it!

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

Now 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>

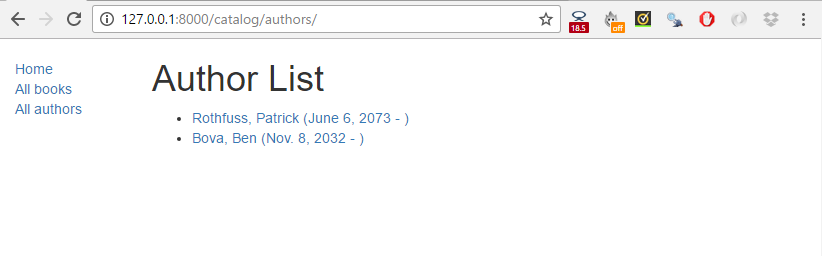
The code required for the URL mappers and the views should be virtually identical to the Book list and detail views we created above. The templates will be different but will share similar behaviour.

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>

Please attempt this on your own before looking at my code. The end result should look something like this:



****

In our next articles, we'll extend this library to support user accounts, and thereby demonstrate user authentication, permissons, sessions, and forms.

For your reference I will now include my files (I have only included the files that I have modified or made during this section, so assume the rest is untouched).

~django\_project/locallibrary/catalog/urls.py

from django.urls import path

from . import views

urlpatterns=[

path('', views.index, name='index'),

path('books/', views.BookListView.as\_view(), name='books'),

path('book/<int:pk>', views.BookDetailView.as\_view(), name='book-detail'),

path('authors/', views.AuthorListView.as\_view(), name='authors'),

path('author/<int:pk>', views.AuthorDetailView.as\_view(), name='author-detail'),

]

~django\_project/locallibrary/catalog/

from django.shortcuts import render

# Create your views here.

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

def index(request):

"""View function for home page of site."""

"""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 copies of books

num\_instances\_available = BookInstance.objects.filter(status\_\_exact='a').count()

num\_authors = Author.objects.count() # The 'all()' is implied by default.

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

from django.views import generic

class BookListView(generic.ListView):

model=Book

paginate\_by=10

class BookDetailView(generic.DetailView):

model=Book

class AuthorListView(generic.ListView):

"""Generic class-based list view for a list of authors."""

model = Author

paginate\_by = 10

class AuthorDetailView(generic.DetailView):

"""Generic class-based detail view for an author."""

model = Author

~django\_project/locallibrary/catalog/templates/index.html

{% extends "base\_generic.html" %}

{% block title %}<title>Local Library Home Page</title>{% endblock %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary</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 %}

~django\_project/locallibrary/catalog/templates/base\_generic.html

<!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://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<!-- 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="{% url 'books' %}">All books</a></li>

<li><a href="{% url 'authors' %}">All authors</a></li>

</ul>

{% endblock %}

</div>

<div class="col-sm-10 ">{% 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">

<p>Page {{ page\_obj.number }} of {{ page\_obj.paginator.num\_pages }}.</p>

</span>

{% if page\_obj.has\_next %}

<a href="{{ request.path }}?page={{ page\_obj.next\_page\_number }}">next</a>

{% endif %}

</span>

</div>

{% endif %}

{% endblock %}

</div>

</div>

</div>

</body>

</html>

~django\_project/locallibrary/catalog/templates/catalog/author\_detail.html

if you do not know what the dl, dd and dt tags are look at this quick example:

https://www.w3schools.com/tags/tryit.asp?filename=tryhtml\_dd\_test

{% extends "base\_generic.html" %}

{% block content %}

<h1>Author: {{ author }} </h1>

<p>{{author.date\_of\_birth}} - {% if author.date\_of\_death %}{{author.date\_of\_death}}{% endif %}</p>

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

<h4>Books</h4>

<dl>

{% for book in author.book\_set.all %}

<dt><a href="{% url 'book-detail' book.pk %}">{{book}}</a> ({{book.bookinstance\_set.all.count}})</dt>

<dd>{{book.summary}}</dd>

{% endfor %}

</dl>

</div>

{% endblock %}

~django\_project/locallibrary/catalog/templates/catalog/author\_list.html

{% extends "base\_generic.html" %}

{% block content %}

<h1>Author List</h1>

{% if author\_list %}

<ul>

{% for author in author\_list %}

<li>

<a href="{{ author.get\_absolute\_url }}">

{{ author }} ({{author.date\_of\_birth}} - {% if author.date\_of\_death %}{{author.date\_of\_death}}{% endif %})

</a>

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no authors available.</p>

{% endif %}

{% endblock %}

~django\_project/locallibrary/catalog/templates/catalog/book\_detail.html

{% 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 == 'm' %}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 %}

~django\_project/locallibrary/catalog/templates/catalog/book\_list.html

{% 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 %}

# Sessions

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 shows how you can use the session framework to provide persistent behaviour for anonymous users in your own sites.

Before we begin I highly encourage you to see this video, it is a good demo of what session is: https://www.youtube.com/watch?v=64veb6tKTm0

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 customized 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.

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.

Here is a nice simple article on what cookies are and what is the pros and cons on them: <http://www.bbc.co.uk/webwise/guides/about-cookies>

We actually created sessions in the second section of this tutorial. The skeleton website actually did have sessions. 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',

....

Go ahead and check this for youself, open up locallibrary/locallibrary/settings.py and see what is in there in the installed apps and middleware lists.

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).

Please get comfortable with python dictionaries if you are not, using this link: <https://www.w3schools.com/python/python_dictionaries.asp>

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 : <https://docs.djangoproject.com/en/2.1/topics/http/sessions/>

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

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).

Okey let us have a simple example, let us just record the visit counts. In this 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

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

'num\_visits': num\_visits,

}

# Render the HTML template index.html with the data in the context variable.

return render(request, 'index.html', context=context)

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 (that is why we modified the context list that we had before, by adding the ‘num\_visits’ to it).

Side note: We might also test whether cookies are even supported in the browser here,or design our UI (user interface or the html) so that it doesn't matter whether or not cookies are supported.

To take the first approach we then need to use something called a test cookie. [Note this is out of the scope of a beginner, but in real life necessary as not all browsers allow cookies to be saved]

Say on the index page (view function of this page) we produce the test cookie using “request.session.set\_test\_cookie()” where request is the user’s request to see the index page.

Then on the immedieately next page so for example the login page we test whether or not this test cookie was saved on the browser or not (we cannot test it on index page). In the view function of the login page we do “if(request.session.test\_cookie\_worked()==True)” then for exmaple we use another cookie to save the user’s username when they log in. See this for an example:

<https://docs.djangoproject.com/en/2.1/topics/http/sessions/#setting-test-cookies>

The other way is to not show the number of views all together, or whether or not the user’s username can or cannot be saved on our records, the user gets the same view. In the real world you see that most websites take the previous approach of alarming the user that he or she needs to enable the cookies or swap the browser as the cookies are now an integral part of most websites. (I am sure you see the dialogs for cookies on every website nowadays after the change in regulations)

Add the line below to your main HTML template (/locallibrary/catalog/templates/index.html) at the bottom of the "Dynamic content" section to display the context variable: (please add this before the {% endblock %} at the end of this file)

<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. (go back to where manage.py is use python3 and call its runserver function as we did before while being in the virtual environment)

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

In our next section I’ll explain the authentication and authorization (permission) framework, and show you how to support user accounts.

I have added here all the files I modified in this section for your reference:

~/django\_project/locallibrary/catalog/views.py

from django.shortcuts import render

# Create your views here.

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

def index(request):

"""View function for home page of site."""

"""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 copies of books

num\_instances\_available = BookInstance.objects.filter(status\_\_exact='a').count()

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

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

'num\_visits': num\_visits,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

from django.views import generic

class BookListView(generic.ListView):

model=Book

paginate\_by=10

class BookDetailView(generic.DetailView):

model=Book

class AuthorListView(generic.ListView):

"""Generic class-based list view for a list of authors."""

model = Author

paginate\_by = 10

class AuthorDetailView(generic.DetailView):

"""Generic class-based detail view for an author."""

model = Author

~/django\_project/locallibrary/catalog/templates/index.html

{% extends "base\_generic.html" %}

{% block title %}<title>Local Library Home Page</title>{% endblock %}

{% block content %}

<h1>Local Library Home</h1>

<p>Welcome to LocalLibrary</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>

<p>You have visited this page {{ num\_visits }}{% if num\_visits == 1 %} time

{% else %} times{% endif %}.</p>

{% endblock %}

# User Authentication and Permissions

Django provides an authentication and authorization ("permission") system, built on top of the session framework discussed in the previous section, 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.

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).

OAuth (Open Authorization) is an open standard for token-based authentication and authorization on the Internet. OAuth allows an end user's account information to be used by third-party services, such as Facebook, without exposing the user's password. You can see examples of these almost everywhere nowadays.

In this tutorial, I'll show you how to enable user authentication in the LocalLibrary website, create your own login and logout pages (and for all users) and add permissions to your models (so to limit access for certain group of people, for example those who are not logged in). 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 log in 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.

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

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

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. Note how our database is now storing more than only books, book instances, genres, languages and authors. We are also prepared to store user data.

To see this for yourself please open the file ~/django\_project/locallibrary/locallibrary/settings.py

See that in Installed\_apps list we have “django.contrib.auth” and in Middleware list we have “django.contrib.auth.middleware.AuthenticationMiddleware”.

You already created your first user when we looked at the Django admin site in section 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.

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'

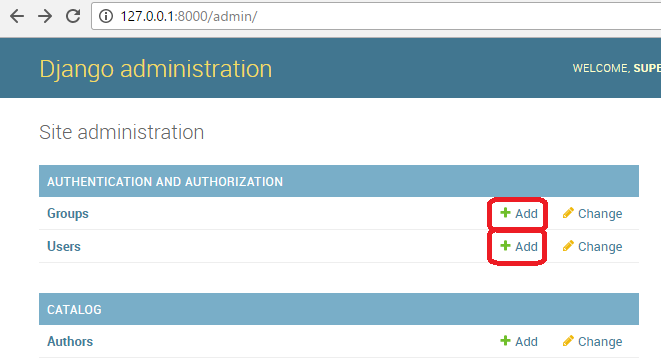
user.last\_name = 'Citizen'

user.save()

You can add such code (above) in views as a views function that is linked to a url for signing up to join the library. So that after submitting a sign up form the data will be passed to this function and then a user account in the database created for the user.

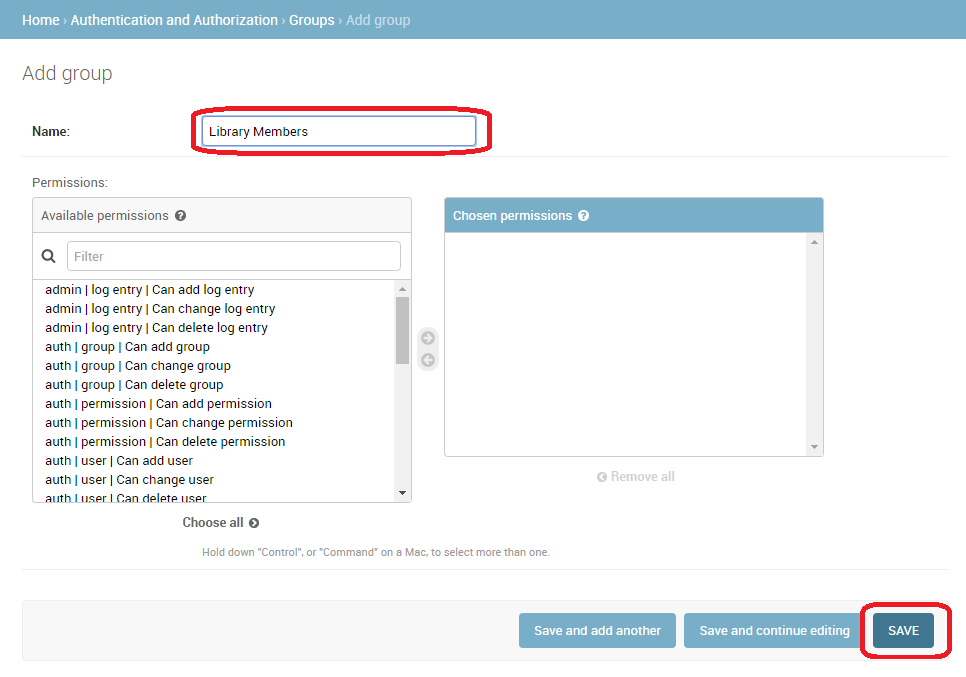
Anyway let us carry on working through this in the admin site instead. Below we'll first create a group and then a user. Even though we don't have any permissions to add for our library members yet, if we need to later, it will be much easier to add them once to the group than individually to each member.

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.

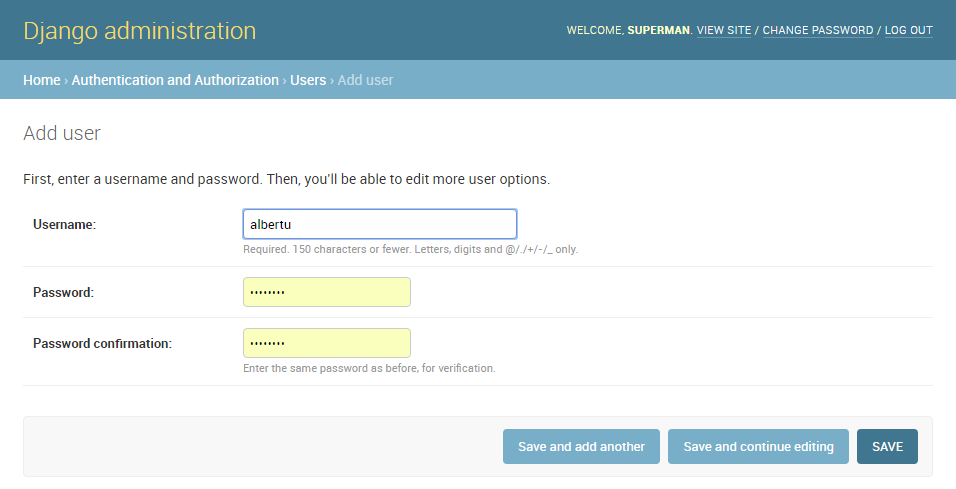


2) We don't need any permissions for the group, so just press SAVE (you will be taken to a list of groups).

3) Now let's create a user:

Navigate back to the home page of the admin site

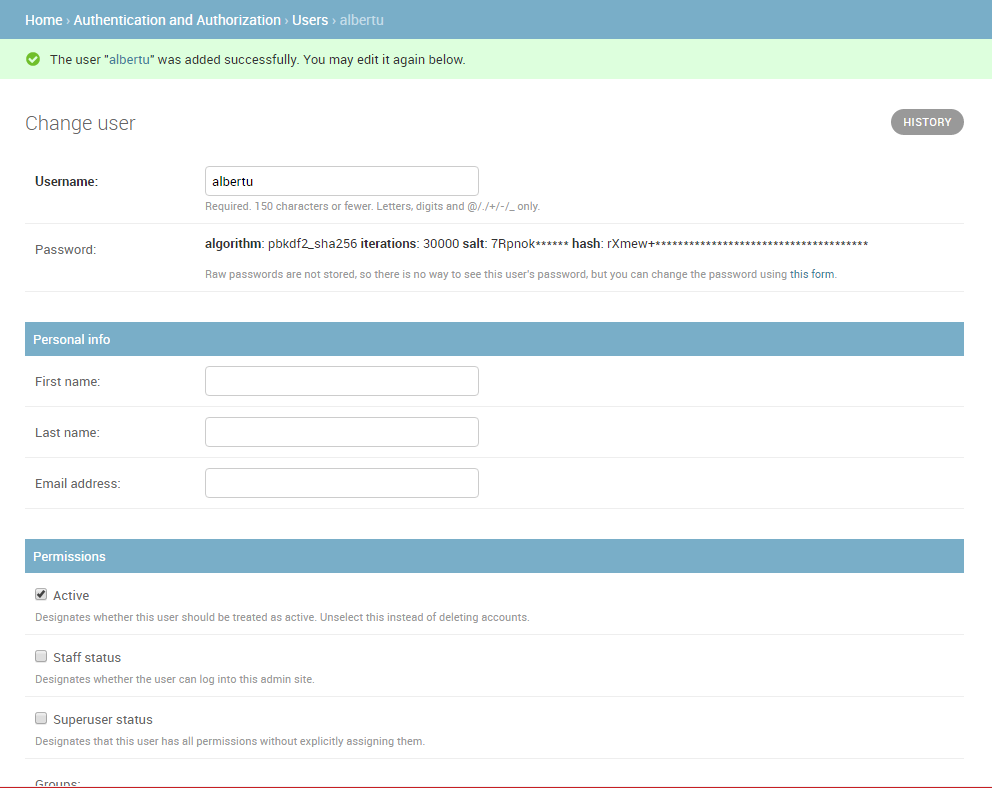
Click the Add button next to Users to open the Add user dialogue.



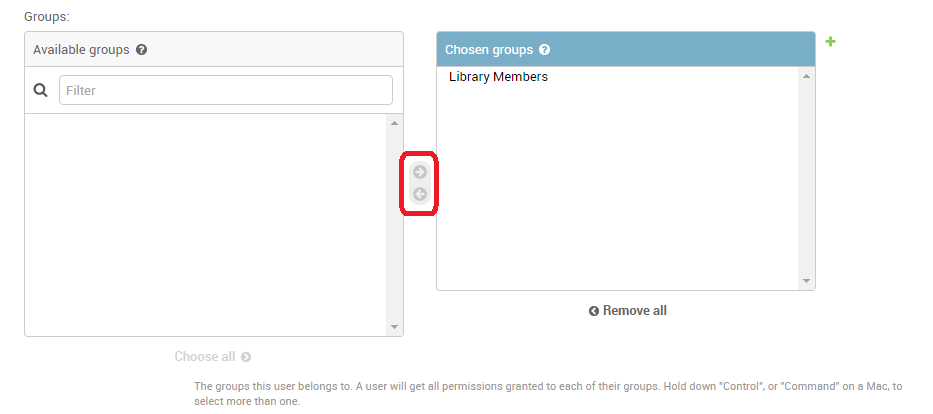
Enter an appropriate Username and Password/Password confirmation for your test user

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, and the user's 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 date).



4) In the Groups section, select Library Member group from the list of Available groups, and then press the right-arrow between the boxes to move it into the Chosen groups box.



We don't need to do anything else here, so just select SAVE again, to go to the list of users.

That's it! 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 log in).

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

Django provides almost everything you need to create authentication pages to handle login, log out, and password management "out of the box". This includes a 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.

Why in the main project and not the catalog application project? Think of Google. We have Gmail, the mailing service of google, Youtube the streamin service owned by Google, GoogleMaps again by Google. For all these applications that are under the main folder of Google, do you have separate accounts, username and passwords? No, as the Google account is shared and is used by YouTube app, GoogleMaps app, and so on. So that is what we are doing here (in case you wanted to make another application as well as the catalog that we already have, then that application would not require the user to make a separate account with it).

Add the following to the bottom of the project urls.py file (locallibrary/locallibrary/urls.py) file: (note again this is not the urls.py file in catalog folder)

#Add Django site authentication urls (for login, logout, password management)

urlpatterns += [

path('accounts/', include('django.contrib.auth.urls')),

]

Run the server (again back in main project folder, using virtual environment and python3 manage.py runserver). Navigate with the browser 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:

admin/

catalog/

^static\/(?P<path>.\*)$

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/<uidb64>/<token>/ [name='password\_reset\_confirm']

accounts/ reset/done/ [name='password\_reset\_complete']

Read the error carefully.

The list above shows paths, or url mappings along with their names in square brackets. So Django has tried to match the url that we put in:

[**http://127.0.0.1:8000/accounts/**](http://127.0.0.1:8000/accounts/)

with any of those (and well no match came up). The first line we remember as it is the admin page, similarly the second one which led us to the index page of the library. The third line is actully the root url of our application which is again same as the index (this was modified in URL intro section). The 4th line is a regular expression (if you are interested to know what these are take a look at: <https://docs.djangoproject.com/en/2.1/topics/http/urls/#using-regular-expressions>, essentially they are a vague term that by some logic can be matched up to an array of choices). The 5th line however points to a good suggestion. That there automatically has been a path generated for us by django named ‘login’. This path can be matched with the url

[**http://127.0.0.1:8000/accounts/login/**](http://127.0.0.1:8000/accounts/login/)

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.

The URLs (and implicitly views) that we just added (believe it or not django literally just created the views and url paths for you, the above url view and path did not exist before we tried to go to account url first!) we are 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 the catalog and locallibrary folders). Please create these folders now (using the mkdir command in terminal).

Your folder structure should now look like the below:

django\_project

|\_locallibrary

|\_catalog

|\_locallibrary

|\_**templates**

|\_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': [os.path.join(BASE\_DIR, 'templates')],

'APP\_DIRS': True,

...

The authentication templates provided in this article are a very basic/slightly modified version of the Django demonstration login templates. You may need to customise them for your own use (when making a real life project, attackers or hackers would be aware of how django by default manages authentication, thus we must vary our code in real life so to make it safer, but at this stage please do not worry about it).

Create a new HTML file called (note the registration folder in its path, make this folder first using mkdir) ~django\_project/locallibrary/templates/registration/login.html and 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 %}

<table>

<tr>

<td>{{ form.username.label\_tag }}</td>

<td>{{ form.username }}</td>

</tr>

<tr>

<td>{{ form.password.label\_tag }}</td>

<td>{{ form.password }}</td>

</tr>

</table>

<input type="submit" value="login" />

<input type="hidden" name="next" value="{{ next }}" />

</form>

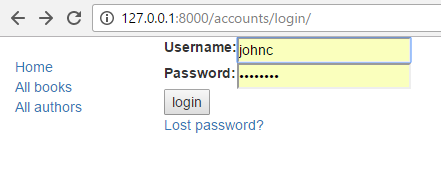
{# Assumes you setup the password\_reset view in your URLconf #}

<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 section. 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 log in 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 log in you should be redirected to the site homepage by default. (try this!)

# Redirect to home URL after login (Default redirects to /accounts/profile/)

LOGIN\_REDIRECT\_URL = '/'

Here is a good time server, in terminal we can have aliases:

For example everytime we need to run the server we first need to make sure we are in the virtual environment so we needed to run this command: ‘workon env’

Then we need to make sure we are in the correct directory where manage.py is so: ‘cd ~/django\_project/locallibrary’

And finally we needed to run: ‘python3 manage.py runserver’

But instead try goin to main user folder so type in terminal:

cd ~

next open the bash profile (we talked about this in section 1). So run this command:

atom .bashrc

Then edit this in atom, add the following line at the bottom:

alias serve=’workon env &&

cd ~/django\_project/locallibrary &&

python3 manage.py runserver’

Next save and close this file. Then refresh the terminal by running this command:

source .bashrc

Now from no matter what directory we are, we can simply type in the terminal:

serve

And the end result is that we are in correct directory, in virtualenvironment, and running the server all at once. Try to get in the habit of using aliases fo tideous and repetetive jobss like this.

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 which by the way is visible if you go to admin site, then click on users, here we see a column dedicated to showing what users have this permission, clearly the admin has it!).

Create and open /locallibrary/templates/registration/logged\_out.html. Copy in the text below:

{% extends "base\_generic.html" %}

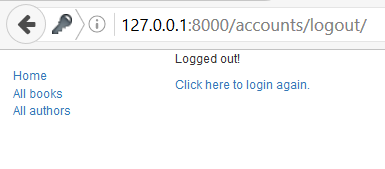
{% block content %}

<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:



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.

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 %}

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 %}

This template provides the text of the HTML email containing the reset link that we will send to users. 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 %}

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">

{% csrf\_token %}

<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 %}

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 %}

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 log in and then log out 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!

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 or the terminal).

EMAIL\_BACKEND = 'django.core.mail.backends.console.EmailBackend'

So essentially try logging in using the link above (accounts/login) then click on reset password, type in the email associated with an existing account or maybe the admin superuser account, then submit, then go to terminal to see this has actually been sent to terminal, copy pase this link (using right click copy or ctrl-shift-c) from terminal and paste it in browser to view, this should ask for a new password, then by submitting it we will be asked to log in again, this time try to login first with the old password to confirm it fails then with the new password.

If you like to know how to actually solve this issue look at: https://docs.djangoproject.com/en/2.1/topics/email/

This part, we look at what we can do to selectively control content the user sees based on whether they are logged in or not.

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 in bold 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.

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.

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):

...

You can do the same sort of thing manually by testing on request.user.is\_authenticated in an if statement in the actual body of the function (consequently manually redirecting the user and setting the next variable), but the decorator is much more convenient!

(Recall: Open views.py in catalog folder, observe that index is a function based view where as BookListView is a class based view as previously discussed in this tutorial.)

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 superclass 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'

Now that we know how to restrict a page to a particular user, let's 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.

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, a user can borrow many bookinstance wheras a book instance can only be borrowed by a user at a given time) 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.

Add this somewhere near the top of the file:

from datetime import date

Now add the following property definition to the BookInstance class:

@property

def is\_overdue(self):

if self.due\_back and date.today() > self.due\_back:

return True

return False

We first verify whether due\_back is empty before making a comparison. An empty due\_back field would cause Django to throw an error instead of showing the page: empty values are not comparable. This is not something we would want our users to experience

Now that we've updated our models, we'll need to make fresh migrations on the project and then apply those migrations: (making sure we are in virtualenv and in correct directory)

python3 manage.py makemigrations

python3 manage.py migrate

Recall we can make aliases for these two commands (if you like).

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, allowing us to 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')

        }),

    )

Now that it's possible to loan books to a specific user, go and loan out a number of BookInstance records (in the admin site). 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: go to the book instance list in admin site, for some of the books listed here, open them, edit the borrower field, set it to a user, change the due back time, and the status. Then save them.

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.

Now open /catalog/urls.py and add a path() pointing to the above view (you can just copy the text below to the end of the file).

urlpatterns += [

path('mybooks/', views.LoanedBooksByUserListView.as\_view(), name='my-borrowed'),

]

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 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).

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>

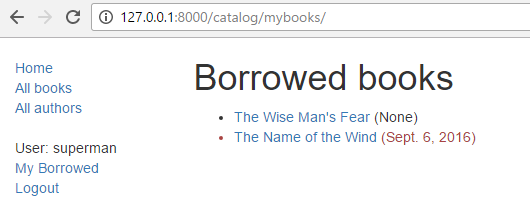
{% else %}

<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, or more precisely I intentionally did not give it one, which is a bug, why? We do not want to loan out books with no due back dates, so my action should not have been permitted in the system even though as an admin of the site, we hope to fix in a later section!).



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).

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: (by the way this is a tutorial on what tuples are in python: https://www.w3schools.com/python/python\_tuples.asp)

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.

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 %

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):

...

A 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 permissions

permission\_required = ('catalog.can\_mark\_returned', 'catalog.can\_edit')

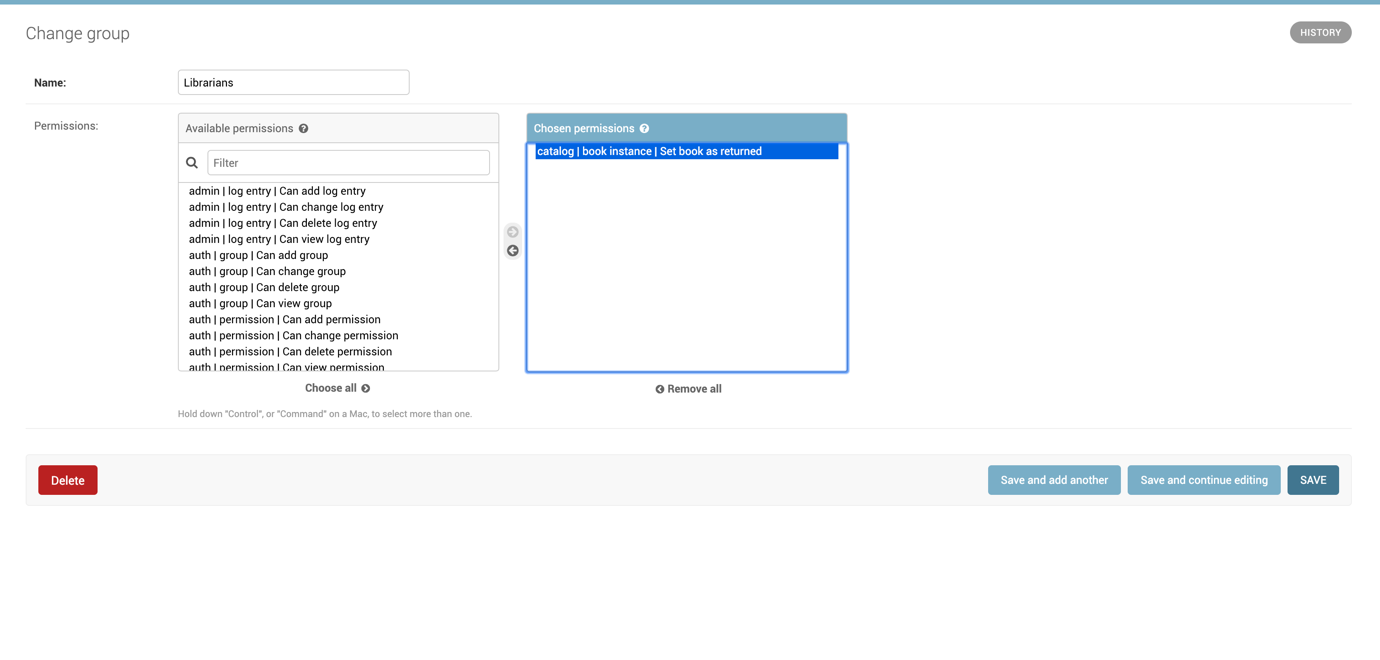
# Note that 'catalog.can\_edit' is just an example

# the catalog application doesn't have such permission!

We won't update the LocalLibrary here; perhaps in the next section!

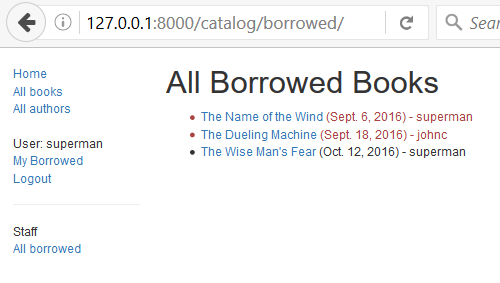
Earlier in this article, we showed you how to create a page for the current user listing the books that they have borrowed. The challenge now is to create a similar page that is only visible for librarians, that displays all books that have been borrowed, and which includes the name of each borrower.

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 I recommend that you instead use the can\_mark\_returned permission (that we added earlier) and PermissionRequiredMixin, as described in the previous section. Remember to give the librarian account the permission (so login the admin site, go to groups, go to librarians, in the permission filter search for "set book as returned" you will find the permission, and then using the arrow symbols next to it, the one pointing right, add it to librarians current permissions, which should be none before you do so, it should look like the image below at the end).



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.

When you are finished, your page should look something like the screenshot below. I have as always copy pasted the files I have modified or created during this section.



~django\_project/locallibrary/catalog/admin.py

from django.contrib import admin

# Register your models here.

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

# We have commented out the ones that are overwritten using admin functions.

#admin.site.register(Book)

admin.site.register(Language)

#admin.site.register(BookInstance)

admin.site.register(Genre)

#admin.site.register(Author)

# Define the admin class

class BooksInline(admin.TabularInline):

"""Defines format of inline book insertion (used in AuthorAdmin) """

model=Book

class AuthorAdmin(admin.ModelAdmin):

"""Administration object for Author models.

Defines:

- fields to be displayed in list view (list\_display)

- orders fields in detail view (fields), grouping the date fields horizontally

- adds inline addition of book in author view (inlines) [scroll to the bottom

of an author page to add books written by him/her]

"""

list\_display=('last\_name','first\_name','date\_of\_birth','date\_of\_death')

fields=['first\_name','last\_name', ('date\_of\_birth', 'date\_of\_death')]

inlines=[BooksInline]

# Register the admin author class with the associated model

admin.site.register(Author, AuthorAdmin)

class BookInstanceInline(admin.TabularInline):

"""Defines format of inline book instance insertion used in (BookAdmin)"""

model = BookInstance

class BookAdmin(admin.ModelAdmin):

"""Administration object for Book models:

Defines:

- fields to be displayed in list view (list\_display)

- adds inline addition of book instances in book view (inlines)

"""

list\_display=('title','author','display\_genre')

inlines = [BookInstanceInline]

# Register the admin book class with associated model

admin.site.register(Book, BookAdmin)

class BookInstanceAdmin(admin.ModelAdmin):

"""Administration object for BookInstace models.

Defines:

- fields to be displayed in list view (list\_display)

- filters that will be displayed in sidebar (list\_filter)

- grouping of fields into sections (fieldsets)

"""

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')

}),

)

# Register the Admin classes for BookInstace

admin.site.register(BookInstance)

~django\_project/locallibrary/catalog/models.py

from django.db import models

from django.contrib.auth.models import User

# Create your models here.

from django.urls import reverse # To generate URLS by reversing URL patterns

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

class Language(models.Model):

"""Model representing a Language (e.g. English, French, Japanese, etc.)"""

name = models.CharField(max\_length=200, help\_text="Enter the book's natural language (e.g. English, French, Japanese etc.)")

def \_\_str\_\_(self):

"""String for representing the Model object (in Admin site etc.)"""

return self.name

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 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 a genre can contain many books and a Book can cover many genres.

# Genre class has already been defined so we can specify the object above.

language = models.ForeignKey('Language', on\_delete=models.SET\_NULL, null=True)

def get\_absolute\_url(self):

"""Returns the url to access a particular book instance."""

return reverse('book-detail', args=[str(self.id)])

def \_\_str\_\_(self):

"""String for representing the Model object."""

return self.title

def display\_genre(self):

"""Create a string for the Genre. This is required to display genre in Admin."""

return ', '.join(genre.name for genre in self.genre.all()[:3])

import uuid # Required for unique book instances

from datetime import date

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)

borrower = models.ForeignKey(User, on\_delete=models.SET\_NULL, null=True, blank=True)

@property

def is\_overdue(self):

if self.due\_back and date.today() > self.due\_back:

return True

return False

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']

permissions=(("can\_mark\_returned","Set book as returned"),)

def \_\_str\_\_(self):

"""String for representing the Model object."""

return '{0} ({1})'.format(self.id, self.book.title)

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)

class Meta:

ordering = ['last\_name', 'first\_name']

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 '{0}, {1}'.format(self.last\_name, self.first\_name)

~django\_project/locallibrary/catalog/templates/base\_generic.html

<!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://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<!-- 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="{% url 'books' %}">All books</a></li>

<li><a href="{% url 'authors' %}">All authors</a></li>

{% 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 %}

{% if perms.catalog.can\_mark\_returned %}

<hr />

<ul class="sidebar-nav">

<li>Staff</li>

<li><a href="{% url 'all-borrowed' %}">All borrowed</a></li>

</ul>

{% endif %}

</ul>

{% endblock %}

</div>

<div class="col-sm-10 ">{% 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">

<p>Page {{ page\_obj.number }} of {{ page\_obj.paginator.num\_pages }}.</p>

</span>

{% if page\_obj.has\_next %}

<a href="{{ request.path }}?page={{ page\_obj.next\_page\_number }}">next</a>

{% endif %}

</span>

</div>

{% endif %}

{% endblock %}

</div>

</div>

</div>

</body>

</html>

~django\_project/locallibrary/catalog/urls.py

from django.urls import path

from . import views

urlpatterns=[

path('', views.index, name='index'),

path('books/', views.BookListView.as\_view(), name='books'),

path('book/<int:pk>', views.BookDetailView.as\_view(), name='book-detail'),

path('authors/', views.AuthorListView.as\_view(), name='authors'),

path('author/<int:pk>', views.AuthorDetailView.as\_view(), name='author-detail'),

path('mybooks/', views.LoanedBooksByUserListView.as\_view(), name='my-borrowed'),

path(r'borrowed/', views.LoanedBooksAllListView.as\_view(), name='all-borrowed'),

]

~django\_project/locallibrary/catalog/views.py

from django.shortcuts import render

# Create your views here.

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

def index(request):

"""View function for home page of site."""

"""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 copies of books

num\_instances\_available = BookInstance.objects.filter(status\_\_exact='a').count()

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

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

'num\_visits': num\_visits,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

from django.views import generic

class BookListView(generic.ListView):

model=Book

paginate\_by=10

class BookDetailView(generic.DetailView):

model=Book

class AuthorListView(generic.ListView):

"""Generic class-based list view for a list of authors."""

model = Author

paginate\_by = 10

class AuthorDetailView(generic.DetailView):

"""Generic class-based detail view for an author."""

model = Author

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')

from django.contrib.auth.mixins import PermissionRequiredMixin

class LoanedBooksAllListView(PermissionRequiredMixin,generic.ListView):

"""Generic class-based view listing all books on loan. Only visible to users with can\_mark\_returned permission."""

model = BookInstance

permission\_required = 'catalog.can\_mark\_returned'

template\_name ='catalog/bookinstance\_list\_borrowed\_all.html'

paginate\_by = 10

def get\_queryset(self):

return BookInstance.objects.filter(status\_\_exact='o').order\_by('due\_back')

~django\_project/locallibrary/locallibrary/settings.py

"""

Django settings for locallibrary project.

Generated by 'django-admin startproject' using Django 2.1.4.

For more information on this file, see

https://docs.djangoproject.com/en/2.1/topics/settings/

For the full list of settings and their values, see

https://docs.djangoproject.com/en/2.1/ref/settings/

"""

import os

# Build paths inside the project like this: os.path.join(BASE\_DIR, ...)

BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/2.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = '8wm5sw$-+kv57b2f)blyai)$tiya\*$5p00ztt#0t0g#((qx@l@'

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'catalog.apps.CatalogConfig'

]

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'locallibrary.urls'

# Note that we have added the directorys of templates to be locallibrary/catalog/templates

# Django needs to be directed to specifically to find these.

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [os.path.join(BASE\_DIR, 'templates')],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

WSGI\_APPLICATION = 'locallibrary.wsgi.application'

# Database

# https://docs.djangoproject.com/en/2.1/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

# Password validation

# https://docs.djangoproject.com/en/2.1/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

{

'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

},

]

# Internationalization

# https://docs.djangoproject.com/en/2.1/topics/i18n/

LANGUAGE\_CODE = 'en-us'

TIME\_ZONE = 'Europe/London'

USE\_I18N = True

USE\_L10N = True

USE\_TZ = True

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/2.1/howto/static-files/

STATIC\_URL = '/static/'

LOGIN\_REDIRECT\_URL ='/'

EMAIL\_BACKEND='django.core.mail.backends.console.EmailBackend'

~django\_project/locallibrary/locallibrary/urls.py

"""locallibrary URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:

https://docs.djangoproject.com/en/2.1/topics/http/urls/

Examples:

Function views

1. Add an import: from my\_app import views

2. Add a URL to urlpatterns: path('', views.home, name='home')

Class-based views

1. Add an import: from other\_app.views import Home

2. Add a URL to urlpatterns: path('', Home.as\_view(), name='home')

Including another URLconf

1. Import the include() function: from django.urls import include, path

2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))

"""

from django.contrib import admin

from django.urls import path

urlpatterns = [

path('admin/', admin.site.urls),

]

#Use include() to add paths from catalog application

from django.urls import include

from django.urls import path

urlpatterns+=[

path('catalog/', include('catalog.urls')),

]

#Add URL maps to redirect the base URL to our application

from django.views.generic import RedirectView

urlpatterns+=[

path('',RedirectView.as\_view(url='/catalog/',permanent=True)),

]

#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)

#Add Django site authentication urls (for login, logout, password management)

urlpatterns+=[

path('accounts/', include('django.contrib.auth.urls')),

]

~django\_project/locallibrary/catalog/templates/catalog/bookinstance\_list\_borrowed\_all.html

Note this was created as part of the challenge

{% extends "base\_generic.html" %}

{% block content %}

<h1>All 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 }}) {% if user.is\_staff %}- {{ bookinst.borrower }}{% endif %}

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no books borrowed.</p>

{% endif %}

{% endblock %}

~django\_project/locallibrary/catalog/templates/catalog/bookinstance\_list\_borrowed\_user.html

{% 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 %}

And the files:

~django\_project/locallibrary/template/registration/logged\_out.html

~django\_project/locallibrary/template/registration/login.html

~django\_project/locallibrary/template/registration/password\_reset\_complete.html

~django\_project/locallibrary/template/registration/password\_reset\_confirm.html

~django\_project/locallibrary/template/registration/password\_reset\_done.html

~django\_project/locallibrary/template/registration/password\_reset\_email.html

~django\_project/locallibrary/template/registration/password\_reset\_form.html

listed in order below:

{% extends "base\_generic.html" %}

{% block content %}

<p>Logged out!</p>

<a href="{% url 'login' %}">Click here to login again.</a>

{% endblock %}

{% 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 %}

{% extends "base\_generic.html" %}

{% block content %}

<h1>The password has been changed!</h1>

<p><a href="{% url 'login' %}">log in again?</a></p>

{% endblock %}

{% extends "base\_generic.html" %}

{% block content %}

{% if validlink %}

<p>Please enter (and confirm) your new password.</p>

<form action="" method="post">

{% csrf\_token %}

<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 %}

{% 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 %}

Someone asked for password reset for email {{ email }}. Follow the link below:

{{ protocol}}://{{ domain }}{% url 'password\_reset\_confirm' uidb64=uid token=token %}

{% 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 %}

I know this chapter was too long, but hopefully you've now created a website that library members can log in 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.

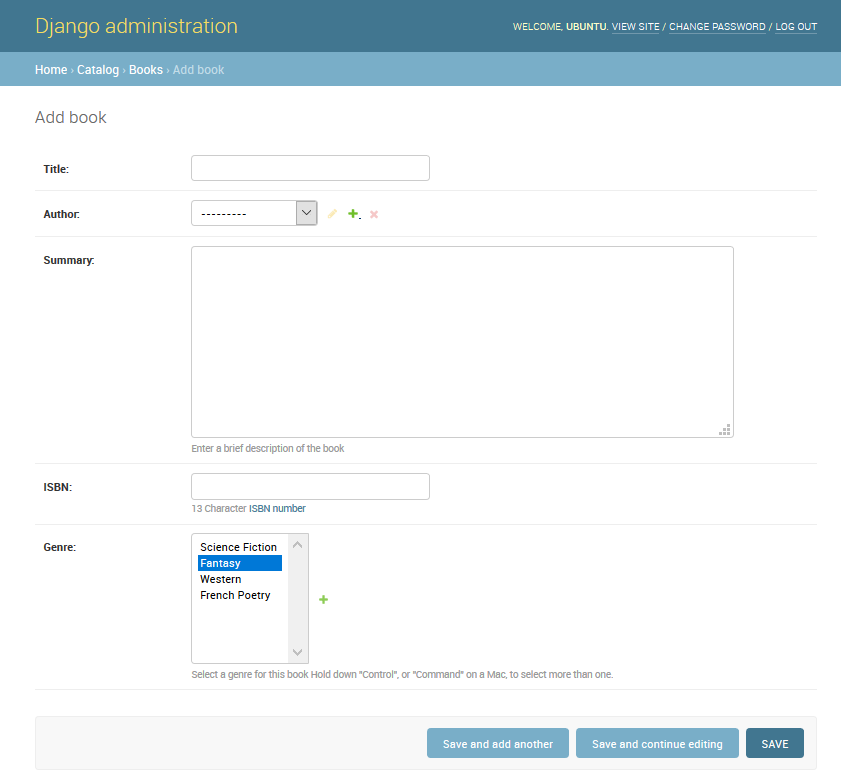
In our next article, we'll look at how you can use Django forms to collect user input, and then start modifying some of our stored data.

# Working with forms

In this section I'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).

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. I highly recommend you to read through this to be comfortable with html forms and how they work: <https://www.w3schools.com/html/html_forms.asp>

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 one of our Book models, comprised of a number of selection lists and text editors.

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, I am 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:



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.

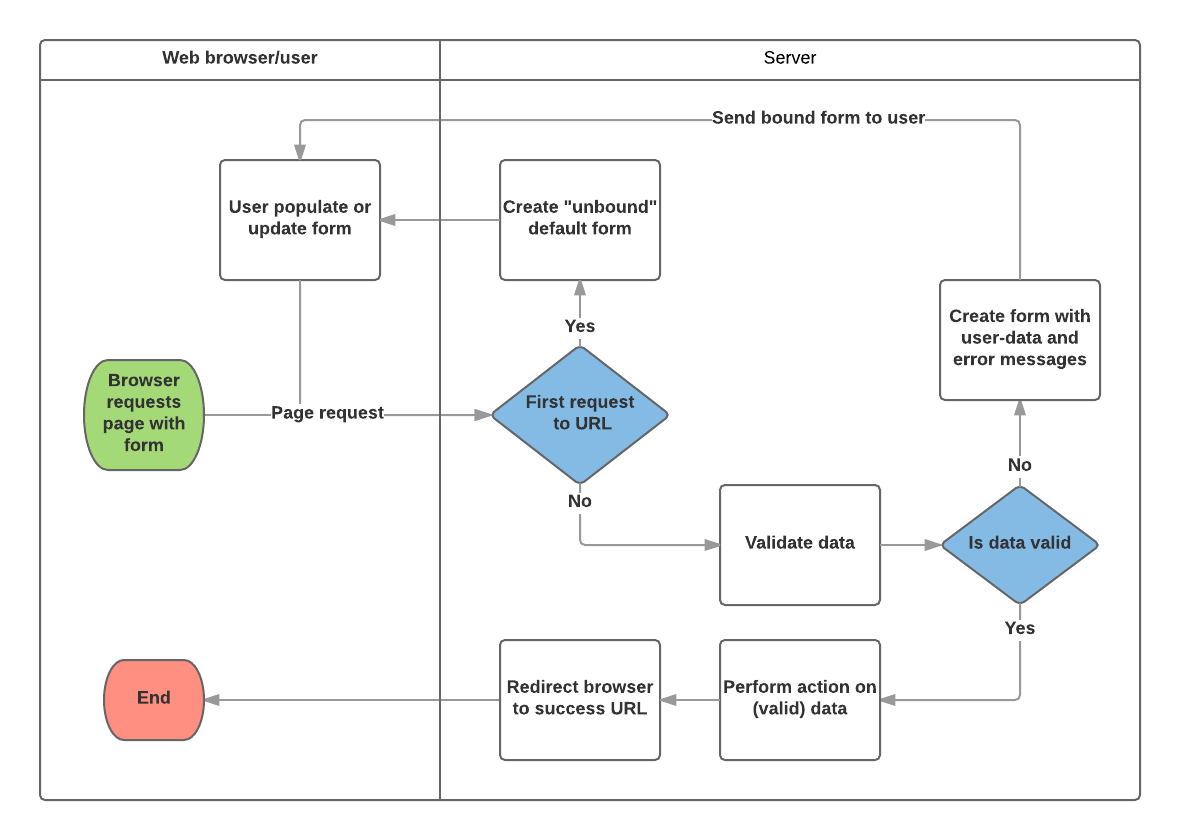
The role of the server is first to render the initial form state — either containing blank fields or pre-populated with initial values. After the user presses the submit button, the server will receive the form data with values from the web browser and must validate the information. If the form contains 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.

Note: what do we mean by valid? Say for example the ISBN field that we might want to add in the form for our own website. The ISBN has to be 13 characters (exactly), so anything bigger or smaller is invalid. You can have similar restrictions for phone number, dates, username passwords and any other field. Evaluation of the values can range from checking the size, the type of inpu (integer or character …) or more sophisticated checks like checking if a username already exists (when creating a new user) in the database which requires we check the database for that username and asking if there is matches already.

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'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 sanitization of the input (e.g. removing invalid characters that might be used to send malicious content to the server) and converts them into consistent Python types.

4)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.)

5)If any data is invalid, re-display the form, this time with any user populated values and error messages for the problem fields.

If all data is valid, perform required actions (e.g. save the data, send and 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. Now I will describe how forms work using the practical example of a page to allow librarians to renew books.

We are 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.

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).

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).")

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 browser widget: DateInput.

There are many other types of form fields, which you will largely recognise from their similarity to the equivalent model field classes, please have a look (scroll down to built-in field classes, they are all listed there):

<https://docs.djangoproject.com/en/2.1/ref/forms/fields/>

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 a label is not specified, Django will create one from the field name by capitalizing 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 a 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 localization of form data input (see <https://docs.djangoproject.com/en/2.1/ref/forms/fields/#localize>

for more information). Do not worry to much about this, but for example think we are getting the price of something from the user. An English user will type 23.50 for 23 pounds and 50 pence, a Dutch user will try to put 23,50 which will not be accepted by the form, and he will consider 2350 as 2 thousand and 350. Localisation will convert the dot to a comma for the Dutch viewer, a different option is to describe to the user what format we are expecting and what it means.

disabled: The field is displayed but its value cannot be edited if this is True. The default is False.

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.

import datetime

from django import forms

from django.core.exceptions import ValidationError

from django.utils.translation import ugettext\_lazy as \_

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 if a date is not in the past.

        if data < datetime.date.today():

            raise ValidationError(\_('Invalid date - renewal in past'))

        # Check if a date is in the allowed range (+4 weeks from today).

        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 sanitized 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 \_() is essentially giving it the nickname “\_”), 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!

Create and open the file locallibrary/catalog/forms.py and copy the entire code listing from the previous block into it.

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 += [

path('book/<uuid:pk>/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. The pattern only matches if pk is a correctly formatted uuid (universally unique id, as described before in previous chapters).

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!

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, we can simply make a new input box with an initial value say 0, this is not displayed by making the CSS style to hidden, then after submission we can change this to 1, after resubmission the value will clearly be 1, and else we know it was not submitted before, this is easily implementable).

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.

import datetime

from django.shortcuts import render, get\_object\_or\_404

from django.http import HttpResponseRedirect

from django.urls import reverse

from catalog.forms import RenewBookForm

def renew\_book\_librarian(request, pk):

    book\_instance = get\_object\_or\_404(RenewBookForm, 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\_instance.due\_back = form.cleaned\_data['renewal\_date']

            book\_instance.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})

context = {

'form': form,

'book\_instance': book\_instance,

}

    return render(request, 'catalog/book\_renew\_librarian.html', context)

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). The HTTP code 302 is exactly the opposite of 404 code.

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\_instance = 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})

context = {

'form': form,

'book\_instance': book\_instance,

}

    return render(request, 'catalog/book\_renew\_librarian.html', context)

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.

However, if 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\_instance = 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\_instance.due\_back = form.cleaned\_data['renewal\_date']

            book\_instance.save()

            # redirect to a new URL:

            return HttpResponseRedirect(reverse('all-borrowed') )

context = {

'form': form,

'book\_instance': book\_instance,

}

    return render(request, 'catalog/book\_renew\_librarian.html', context)

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 sanitized, 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 part of the challenge I gave you in the previous chapter if you do not have this, return the root or index page ‘/’ instead).

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.

import datetime

from django.contrib.auth.decorators import permission\_required

from django.shortcuts import get\_object\_or\_404

from django.http import HttpResponseRedirect

from django.urls import reverse

from catalog.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\_instance = 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\_instance.due\_back = form.cleaned\_data['renewal\_date']

book\_instance.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})

context = {

'form': form,

'book\_instance': book\_instance,

}

return render(request, 'catalog/book\_renew\_librarian.html', context)

Create the template referenced in the view (/catalog/templates/catalog/book\_renew\_librarian.html) and copy the code below into it: (why? Well take a look at the final line in the code above, we need this template)

{% extends "base\_generic.html" %}

{% block content %}

<h1>Renew: {{ book\_instance.book.title }}</h1>

<p>Borrower: {{ book\_instance.borrower }}</p>

<p{% if book\_instance.is\_overdue %} class="text-danger"{% endif %}>Due date: {{ book\_instance.due\_back }}</p>

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</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 {{ book\_instance }} (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 (essentially the code below is what gets rendered to the user’s browser):

<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. 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 (this is automatically generated, but shown below is what can be for example rendered in the user’s browser).

<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>

Using {{ form.as\_table }} 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 }}).

It is also possible to 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.

For more examples of how to manually render forms in templates and dynamically loop over template fields, see:

<https://docs.djangoproject.com/en/2.1/topics/forms/#rendering-fields-manually>

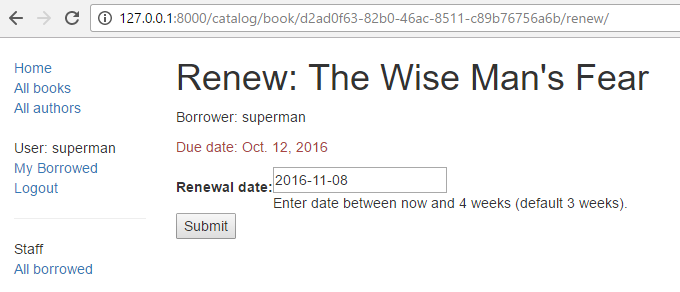
After adding the view all borrowed page for librarians (as we did in the previous chapter), you'll have a list of all books on loan in the library, which is only visible to librarians. We can add a link to our renew page next to each item using the template code below.

So first open up ~django\_project/locallibrary/catalog/templates/catalog/bookinstance\_list\_borrowed\_all.html

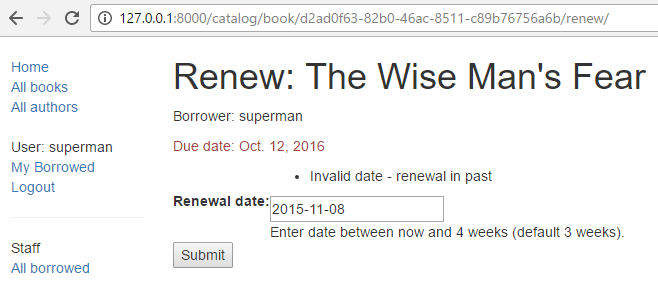
Then add the below code just before the closing tag </li>

{% if perms.catalog.can\_mark\_returned %}- <a href="{% url 'renew-book-librarian' bookinst.id %}">Renew</a> {% endif %}

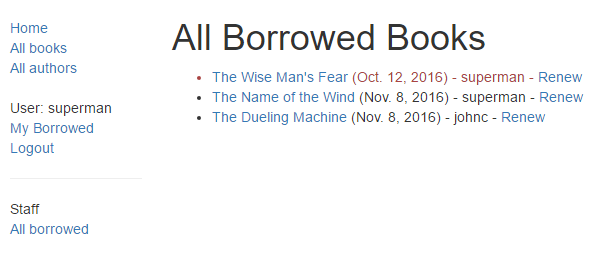
I have provided you at the end of this chapter how this file should look after this modification, you can double checkit there to make sure you have made the right modification. If you are successful, the default form will look like this:



The form with an invalid value entered will look like this:



The list of all books with renew links will look like this:



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 catalog.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 Back), 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': \_('New 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". This change is necessary since the corresponding field in BookInstance is called due\_back. .

from django.forms import ModelForm

from catalog.models import BookInstance

class RenewBookModelForm(ModelForm):

    def clean\_due\_back(self):

      data = self.cleaned\_data['due\_back']

  # Check if a date is not in the past.

       if data < datetime.date.today():

          raise ValidationError(\_('Invalid date - renewal in past'))

      # Check if a date is in the allowed range (+4 weeks from today).

      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 as long as you also update the corresponding form variable name from renewal\_date to due\_back as in the second form declaration: RenewBookModelForm(initial={'due\_back': proposed\_renewal\_date}.

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" behavior, 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 patterns. 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 reimplementation 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).

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 catalog.models import Author

class AuthorCreate(CreateView):

model = Author

fields = '\_\_all\_\_'

initial = {'date\_of\_death': '10/01/2019'}

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, update, or delete 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 the 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 a URL to a class-based view attribute.

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" value="Yes, delete.">

</form>

{% endblock %}

Open your URL configuration file (locallibrary/catalog/urls.py) and add the following configuration to the bottom of the file:

urlpatterns += [

    path('author/create/', views.AuthorCreate.as\_view(), name='author\_create'),

    path('author/<int:pk>/update/', views.AuthorUpdate.as\_view(), name='author\_update'),

    path('author/<int:pk>/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 recognize 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: We didn't do anything to prevent unauthorized 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).

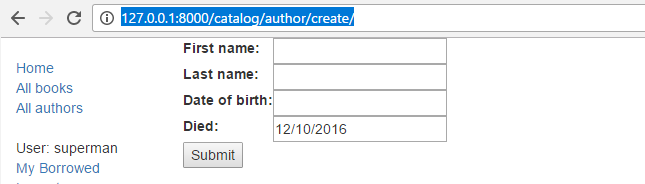
I will show you my answer at the end of this chapter.

Let us test it. Run the server, then login as a non Librarian (so without the permission talked about before). Then try to see the page: <http://127.0.0.1:8000/catalog/author/10/update/> where 10 is the id of an author that already exists in your site (to check what authors you have, go to admin site, select and author, when looking at the details of the author, pay attention to the url of the page, the url should be something like .../admin/catalog/author/2/change, the number 2 is the id of this author).

When logged in as a non-Librarian, we should get an error sayin “Forbidden page” correctly as the user does not have the permission to view this page.

Then, log in to the site with an account that has whatever permissions you decided are needed to access the author editing pages. (mines is set book as reruned because I did not change it from the previous chapter)

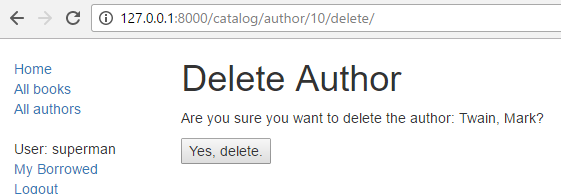
Then navigate to the author create page: http://127.0.0.1:8000/catalog/author/create/, which should look like the screenshot below.



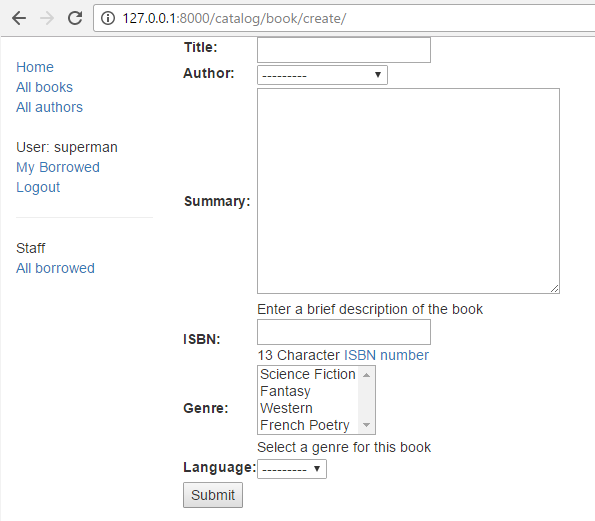
Enter values for the fields and then press Submit to save the author record. You should now be taken to a detail view for your new author, with a URL of something like http://127.0.0.1:8000/catalog/author/10.

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!

Finally, 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.



As a final challenge for this section, 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:



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.

Now I provide the files that I have added or modified during this chapter:

~/django\_project/locallibrary/catalog/templates/catalog/bookinstance\_list\_borrowed\_all.html

{% extends "base\_generic.html" %}

{% block content %}

<h1>All 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 }}) {% if user.is\_staff %}- {{ bookinst.borrower }}{% endif %}

{% if perms.catalog.can\_mark\_returned %}- <a href="{% url 'renew-book-librarian' bookinst.id %}">Renew</a> {% endif %}

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no books borrowed.</p>

{% endif %}

{% endblock %}

~/django\_project/locallibrary/catalog/urls.py

from django.urls import path

from . import views

urlpatterns=[

path('', views.index, name='index'),

path('books/', views.BookListView.as\_view(), name='books'),

path('book/<int:pk>', views.BookDetailView.as\_view(), name='book-detail'),

path('authors/', views.AuthorListView.as\_view(), name='authors'),

path('author/<int:pk>', views.AuthorDetailView.as\_view(), name='author-detail'),

path('mybooks/', views.LoanedBooksByUserListView.as\_view(), name='my-borrowed'),

path('borrowed/', views.LoanedBooksAllListView.as\_view(), name='all-borrowed'),

path('book/<uuid:pk>/renew/', views.renew\_book\_librarian, name='renew-book-librarian'),

path('author/create/', views.AuthorCreate.as\_view(), name='author\_create'),

path('author/<int:pk>/update/', views.AuthorUpdate.as\_view(), name='author\_update'),

path('author/<int:pk>/delete/', views.AuthorDelete.as\_view(), name='author\_delete'),

path('book/create/', views.BookCreate.as\_view(), name='book\_create'),

path('book/<int:pk>/update/', views.BookUpdate.as\_view(), name='book\_update'),

path('book/<int:pk>/delete/', views.BookDelete.as\_view(), name='book\_delete'),

]

~/django\_project/locallibrary/catalog/views.py

from django.shortcuts import render

# Create your views here.

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

def index(request):

"""View function for home page of site."""

"""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 copies of books

num\_instances\_available = BookInstance.objects.filter(status\_\_exact='a').count()

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

context = {

'num\_books': num\_books,

'num\_instances': num\_instances,

'num\_instances\_available': num\_instances\_available,

'num\_authors': num\_authors,

'num\_visits': num\_visits,

}

# Render the HTML template index.html with the data in the context variable

return render(request, 'index.html', context=context)

from django.views import generic

class BookListView(generic.ListView):

model=Book

paginate\_by=10

class BookDetailView(generic.DetailView):

model=Book

class AuthorListView(generic.ListView):

"""Generic class-based list view for a list of authors."""

model = Author

paginate\_by = 10

class AuthorDetailView(generic.DetailView):

"""Generic class-based detail view for an author."""

model = Author

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')

from django.contrib.auth.mixins import PermissionRequiredMixin

class LoanedBooksAllListView(PermissionRequiredMixin,generic.ListView):

"""Generic class-based view listing all books on loan. Only visible to users with can\_mark\_returned permission."""

model = BookInstance

permission\_required = 'catalog.can\_mark\_returned'

template\_name ='catalog/bookinstance\_list\_borrowed\_all.html'

paginate\_by = 10

def get\_queryset(self):

return BookInstance.objects.filter(status\_\_exact='o').order\_by('due\_back')

#added as a part of the 'forms' chapter

import datetime

from django.contrib.auth.decorators import permission\_required

from django.shortcuts import get\_object\_or\_404

from django.http import HttpResponseRedirect

from django.urls import reverse

from catalog.forms import RenewBookForm

#can change this later to 'can\_renew'

@permission\_required('catalog.can\_mark\_returned')

def renew\_book\_librarian(request, pk):

"""View function for renewing a specific BookInstance by librarian."""

book\_instance = 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\_instance.due\_back = form.cleaned\_data['renewal\_date']

book\_instance.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})

context = {

'form': form,

'book\_instance': book\_instance,

}

return render(request, 'catalog/book\_renew\_librarian.html', context)

from django.views.generic.edit import CreateView, UpdateView, DeleteView

from django.urls import reverse\_lazy

from catalog.models import Author

class AuthorCreate(PermissionRequiredMixin, CreateView):

model = Author

fields = '\_\_all\_\_'

initial = {'date\_of\_death': '10/01/2019'}

permission\_required = 'catalog.can\_mark\_returned'

class AuthorUpdate(PermissionRequiredMixin, UpdateView):

model = Author

fields = ['first\_name', 'last\_name', 'date\_of\_birth', 'date\_of\_death']

permission\_required = 'catalog.can\_mark\_returned'

class AuthorDelete(PermissionRequiredMixin, DeleteView):

model = Author

success\_url = reverse\_lazy('authors')

permission\_required = 'catalog.can\_mark\_returned'

class BookCreate(PermissionRequiredMixin, CreateView):

model = Book

fields = '\_\_all\_\_'

permission\_required = 'catalog.can\_mark\_returned'

class BookUpdate(PermissionRequiredMixin, UpdateView):

model = Book

fields = '\_\_all\_\_'

permission\_required = 'catalog.can\_mark\_returned'

class BookDelete(PermissionRequiredMixin, DeleteView):

model = Book

success\_url = reverse\_lazy('books')

permission\_required = 'catalog.can\_mark\_returned'

~/django\_project/locallibrary/catalog/forms.py

import datetime

from django import forms

from django.core.exceptions import ValidationError

from django.utils.translation import ugettext\_lazy as \_

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 if a date is not in the past.

if data < datetime.date.today():

raise ValidationError(\_('Invalid date - renewal in past'))

# Check if a date is in the allowed range (+4 weeks from today).

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

~/django\_project/locallibrary/catalog/template/catalog/author\_form.html

{% extends "base\_generic.html" %}

{% block content %}

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</table>

<input type="submit" value="Submit">

</form>

{% endblock %}

~/django\_project/locallibrary/catalog/template/catalog/author\_confirm\_delete.html

{% 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" value="Yes, delete.">

</form>

{% endblock %}

~/django\_project/locallibrary/catalog/template/catalog/book\_renew\_librarian.html

{% extends "base\_generic.html" %}

{% block content %}

<h1>Renew: {{ book\_instance.book.title }}</h1>

<p>Borrower: {{ book\_instance.borrower }}</p>

<p{% if book\_instance.is\_overdue %} class="text-danger"{% endif %}>Due date: {{ book\_instance.due\_back }}</p>

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</table>

<input type="submit" value="Submit">

</form>

{% endblock %}

~/django\_project/locallibrary/catalog/template/catalog/book\_form.html

{% extends "base\_generic.html" %}

{% block content %}

<form action="" method="post">

{% csrf\_token %}

<table>

{{ form.as\_table }}

</table>

<input type="submit" value="Submit" />

</form>

{% endblock %}

~/django\_project/locallibrary/catalog/template/catalog/book\_confirm\_delete.html

{% extends "base\_generic.html" %}

{% block content %}

<h1>Delete Book</h1>

<p>Are you sure you want to delete the book: {{ book }}?</p>

<form action="" method="POST">

{% csrf\_token %}

<input type="submit" action="" value="Yes, delete." />

</form>

{% endblock %}

# Testing a Django 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. I will first invite you to read through this tutorial showing how unit testing is perfomred in python in general before we talk about django:

<https://docs.python-guide.org/writing/tests/>

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 chapter). 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 the 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 chapter shows how to write automated tests for Django, by adding a number of tests to the LocalLibrary website.

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 us? 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 (check out this video, it is at a very basic level how this library works <https://www.youtube.com/watch?v=6tNS--WetLI>). 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. Check this link if you are interested to know: <https://docs.djangoproject.com/en/2.1/topics/testing/tools/#liveservertestcase>

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.

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.

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 let's 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. You can create the init file by running this command while inside the tests direcotry:

touch \_\_init\_\_.py

copying the tests.py file from catalog folder to the tests folder inside catalog can be done like so: (while in catalog folder to begin with)

mv tests.py tests

to copy the tests.py file and change its name inside the tests directory try:

cp tests.py test\_views.py

similarly try the above line with forms and models instead of views.

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.

Deleting a file is done with this command:

rm tests.py

Open /catalog/tests/test\_models.py. The file should import 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).

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.

If you get errors similar to: ValueError: Missing staticfiles manifest entry ... this may be because testing does not run collectstatic by default and your app is using a storage class that requires it (see https://docs.djangoproject.com/en/2.1/ref/contrib/staticfiles/#django.contrib.staticfiles.storage.ManifestStaticFilesStorage.manifest\_strict" for more information). There are a number of ways you can overcome this problem - the easiest is to simply run collectstatic before running the tests:

python3 manage.py collectstatic

Run the tests in the root directory of LocalLibrary. You should see an output like the one below.

> python3 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.**

**setUp: 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.py", 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.

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".

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:

# Run the specified module

python3 manage.py test catalog.tests

# Run the specified module

python3 manage.py test catalog.tests.test\_models

# Run the specified class

python3 manage.py test catalog.tests.test\_models.YourTestClass

# Run the specified method

python3 manage.py test catalog.tests.test\_models.YourTestClass.test\_one\_plus\_one\_equals\_two

Now we know how to run our tests and what sort of things we need to test, let's look at some practical examples. I won't write every possible test, but this should give you an idea of how tests work, and what more you can do.

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 f'{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

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.assertEquals(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.assertEquals(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.assertEquals(max\_length, 100)

def test\_object\_name\_is\_last\_name\_comma\_first\_name(self):

author = Author.objects.get(id=1)

expected\_object\_name = f'{author.last\_name}, {author.first\_name}'

self.assertEquals(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.assertEquals(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:

# Get an author object to test

author = Author.objects.get(id=1)

# Get the metadata for the required field and use it to query the required field data

field\_label = author.\_meta.get\_field('first\_name').verbose\_name

# Compare the value to the expected result

self.assertEquals(field\_label, 'first name')

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. (note verbose name is the human readable format or the value)

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. (I have added it in my final version)

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 = f'{author.last\_name}, {author.first\_name}'

    self.assertEquals(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.assertEquals(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\_death 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).

E NNNNDDDD