**Overview**

1. Use the django-admin tool to create the project folder, basic file templates, and project management script (**manage.py**).
2. Use **manage.py** to create one or more *applications*.
3. Register the new applications to include them in the project.
4. Hook up the url/path mapper for each application.

*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)

## Creating the project

First open a command prompt/terminal, make sure you are in your [virtual environment](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/development_environment#Using_a_virtual_environment), navigate to where you want to store your Django apps (make it somewhere easy to find like inside your *documents* folder), and create a folder for your new website (in this case: *django\_projects*). Then enter into the folder using the cd command:

mkdir django\_projects

cd django\_projects

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

django-admin startproject locallibrary

cd locallibrary

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

*locallibrary/*

  manage.py

*locallibrary/*

\_\_init\_\_.py

    settings.py

    urls.py

    wsgi.py

Our current working directory should look something like this:

../django\_projects/locallibrary/

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

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

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

## Creating the catalog application

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

python3 manage.py startapp catalog

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

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

If you are using Python 3.7.0 or later, you should only use  py manage.py startapp catalog

The updated project directory should now look like this:

*locallibrary/*

  manage.py

*locallibrary/*

***catalog/***

**admin.py**

**apps.py**

**models.py**

**tests.py**

**views.py**

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

***migrations/***

In addition we now have:

* A *migrations* folder, used to store "migrations" — files that allow you to automatically update your database as you modify your models.
* **\_\_init\_\_.py** — an empty file created here so that Django/Python will recognise the folder as a [Python Package](https://docs.python.org/3/tutorial/modules.html#packages) and allow you to use its objects within other parts of the project.

### **6.4.1. Importing \* From a Package**[**¶**](https://docs.python.org/3/tutorial/modules.html#importing-from-a-package)

. For example, the file sound/effects/\_\_init\_\_.py could contain the following code:

\_\_all\_\_ = ["echo", "surround", "reverse"]

This would mean that from sound.effects import \* would import the three named submodules of the sound package.

If \_\_all\_\_ is not defined, the statement from sound.effects import \* does *not* import all submodules from the package sound.effects into the current namespace; it only ensures that the package sound.effects has been imported

## Hooking up the URL mapper

**Note**: The route in path() is a string defining a URL pattern to match. This string might include a named variable (in angle brackets), e.g. 'catalog/<id>/'. This pattern will match a URL like **/catalog/**any\_chars**/** and pass any\_chars to the view as a string with parameter name id. We discuss path methods and route patterns further in later topics.

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

urlpatterns = [

  path('admin/', admin.site.urls),

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

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

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

Running database migrations

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

python3 manage.py makemigrations

python3 manage.py migrate

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

### **Model definition**

Models are usually defined in an application's **models.py** file. They are implemented as subclasses of django.db.models.Model, and can include fields, methods and metadata.

#### Fields

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

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.

##### **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.

##### **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\_length for the field, but this is used only when the field is displayed in forms (it is not enforced at the database level).
* [IntegerField](https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.IntegerField) 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/" \l "datetimefield) are used for storing/representing dates and date/time information (as Python datetime.date in and datetime.datetime objects, respectively). These fields can additionally declare the (mutually exclusive) parameters auto\_now=True (to set the field to the current date every time the model is saved), auto\_now\_add (to only set the date when the model is first created) , and default (to set a default date that can be overridden by the user).
* [EmailField](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/" \l "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" (models containing a "foreign key" referring to that "key", are on the "many" side of such a relationship).
* [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).

#### 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.