Software Requirements.
IDE:pycharam
Django:3 and above
Python:3 and above(preferred 3.9)

1)Open a command shell (or a terminal window), and make sure you are in your virtual environment.

2)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 change into your newly-created directory:

```
mkdir django_projects
cd django_projects
```

3)Create the new project using the `django-admin startproject` command as shown, and then change into the project folder:

```
django-admin startproject Login
cd Login
```

4)The `django-admin` tool creates a folder/file structure as follows:

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

4)Our current working directory should look something like this:

```
../django_projects/Login/
```

5)**__init__.py** is an empty file that instructs Python to treat this directory as a Python package.

- **settings.py** contains all the website settings, including registering 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 mappings to particular applications, as you'll see later.
- **wsgi.py** is used to help your Django application communicate with the webserver. You can treat this as boilerplate.
- **asgi.py** is a standard for Python asynchronous web apps and servers to communicate with each other. ASGI is the asynchronous successor to WSGI and provides a standard for both asynchronous and synchronous Python apps (whereas WSGI provided a standard for synchronous apps only). It is backward-compatible with WSGI and supports multiple servers and application frameworks.

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

6)Next, run the following command to create the *catalog* application that will live inside our *Login* project. Make sure to run this command from the same folder as your project's **manage.py**:
`python3 manage.py startapp users`

7)**Note:** The example command is for Linux/macOS X. On Windows, the command should be:
`py -3 manage.py startapp users`

If you're working on Windows, replace `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 tool creates a new folder and populates it with files for the different parts of the application (shown in the following example). Most of the files are named after their purpose (e.g. views should be stored in **views.py**, models in **models.py**, tests in **tests.py**, administration site configuration in **admin.py**, application registration in **apps.py**) and contain some minimal boilerplate code for working with the associated objects.

The updated project directory should now look like this:

```
Login/
  manage.py
  Login/
  users/
    admin.py
    apps.py
    models.py
    tests.py
    views.py
    __init__.py
    migrations/
```

8)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 recognize the folder as a Python Package and allow you to use its objects within other parts of the project.

9)Registering Users application

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

```
INSTALLED_APPS = [
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Add our new application
     'Users',

]
```

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

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

11)Other project settings

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

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

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

- SECRET_KEY. This is a secret key that is used as part of Django's website security strategy.

  If you're not protecting this code in development, you'll need to use a different code (perhaps read from an environment variable or file) when putting it into production.
- DEBUG. This enables debugging logs to be displayed on error, rather than HTTP status code responses. This should be set to False in production as debug information is useful for attackers, but for now we can keep it set to True.

11)Hooking up the URL mapper

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

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),
]
```

To add a new list item to the urlpatterns list, add the following lines to the bottom of the file. This new item includes a path() that forwards requests with the pattern catalog/ to the module catalog.urls (the file with the relative URL users**/urls.py**).

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

urlpatterns += [
    path('users/', include('users.urls')),
]
```

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

13)Running database migrations

Django uses an Object-Relational-Mapper (ORM) to map model definitions in the Django code to the data structure used by the underlying database. As we change our model definitions, Django tracks the changes and can create database migration scripts (in **/Login/users/migrations/**) to automatically migrate the underlying data structure in the database to match the model.

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

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

The migrate command is what applies the migrations to your database. Django tracks which ones have been added to the current database.

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

```
python3 manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).
August 15, 2018 - 16:11:26
Django version 2.1, using settings 'locallibrary.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

15)View (function-based)
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.

16)A template is a text file that defines the structure or layout of a file (such as an HTML page), it uses placeholders to represent actual content.
A Django application created using **startapp** (like the skeleton of this example) will look for templates in a subdirectory named '**templates**' of your applications

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

17)Creating a superuser

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

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

```
python3 manage.py createsuperuser
```