

Chapter 1: Development of a Django web app

1.1 Introduction

This course will teach you how to develop an interactive, data-driven, website. You have done a similar course in your second year with ASP.Net, but in this course we will expand your horizons with exposure to a new environment.

For the client side, we will use html, css, and JavaScript with which you are well familiar (I trust). For the server-side, we will use Python with the Django framework and a SQLite database. I think that development in Django is easier and quicker than with ASP.net - after you mastered the learning curve. So, hang in there - the rewards are good.

This course focuses on interaction and functionality. We are not as much interested in graphical design, but that is not to say, however, that the user interface should not be neat. The pages will mostly be boring black-on-white without pictures and a variety of fonts and graphical elements, but they will serve the purpose to manage data on server-side.

We will start with the development of a basic Django web-app prior to discussing some Django basics. The running app will allow us to refer to specific examples when discussing the theory.

1.2 Example scenario

The example in the first three chapters will be based on the following scenario: Run an online shop where customers can place orders and print invoices.

1.3 localhost or online?

You have a choice between development environments. You can either do everything online in PythonAnywhere, or you can download and install Python, a Python editor (e.g. VS Code), and a database server on your local PC.

We will follow the localhost route. It is cheaper and debugging is much faster. Once you are sure that your site is running, you can create an account on PythonAnywhere and publish your site. See Appendix E for details.

1.4 Check Python installation

```
$ python --version
```

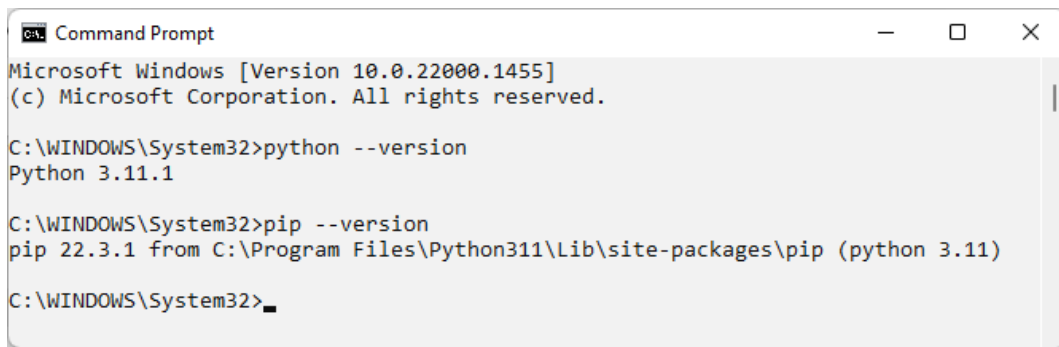
If Python is not installed, download and install from here:

<https://www.python.org/downloads/>.

By the way, the symbol \$ is used as a shortcut for the command prompt. Wherever you see \$, replace with something like C:\Windows\System32>. Of course, the subfolder will depend on the current context.

Check also that the package installer, pip, is working properly:

```
$ pip --version
```



```
Command Prompt
Microsoft Windows [Version 10.0.22000.1455]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\System32>python --version
Python 3.11.1

C:\WINDOWS\System32>pip --version
pip 22.3.1 from C:\Program Files\Python311\Lib\site-packages\pip (python 3.11)

C:\WINDOWS\System32>
```

- References:
 - Python
 - <https://www.python.org/>
 - <https://docs.python.org/3.8/tutorial/index.html>
 - PIP
 - <https://pypi.org/project/pip/>

1.5 Create new web app

1.5.1 Naming conventions

Since we create an app inside a project inside a virtual environment, we prefer to prefix each of these elements to distinguish between the environment, project, and app. Remember that in Python we don't use Pascal case or camel case, but rather all lower case with an underscore between words.

Environment: env_
Project: pr_
App: app_

For production environments, we normally omit the prefix for the app because that is what the end user will see. Please don't do it for this course.

1.5.2 Steps

A Django app resides inside a project, which in turn resides inside a virtual environment. So, there are some steps to follow to create a new app.

(See Appendix D (Section 11.1 and further), for a summary of the steps to create a new Django app.)

- Virtual environment (venv)
- Install Django in the venv
- Create project inside the venv
- Start the server
- Prepare the editor
- Create app
- Configure
- Add an index page
- Test
- foreach other page
 - Create the page
 - Test

1.5.3 Virtual environment

A virtual environment (venv) keeps dependencies that are required for different projects separate. So, when we install a Python library, we do so in the virtual environment so that it is isolated from other projects. Of course, this means that if you want to use the same library for another project, you will have to install it again, but it is considered good Python practice. See this link for more details:

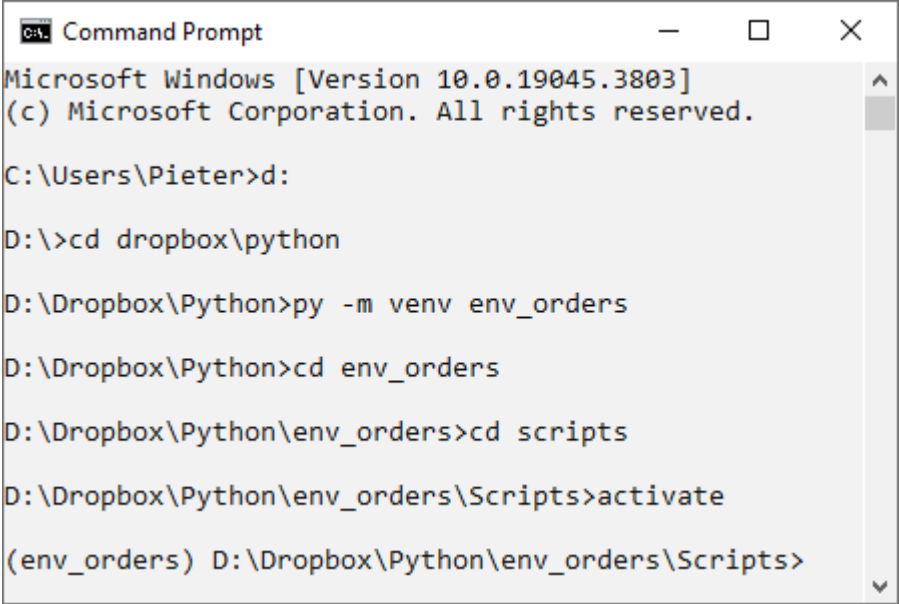
<https://www.geeksforgeeks.org/python-virtual-environment/> .

In a command window, browse to the location where you want to create the environment. On my computer, I created a Python folder where all my Python projects reside.

```
D:\sod517c>py -m venv env_orders
```

Activate. Do every time when you start working. Thereafter, the command prompt will be preceded with the name of the virtual environment in brackets.

```
D:\Dropbox\Python\env_orders\scripts> activate.bat
(env_orders) $
```



```
Command Prompt
Microsoft Windows [Version 10.0.19045.3803]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Pieter>d:

D:\>cd dropbox\python

D:\Dropbox\Python>py -m venv env_orders

D:\Dropbox\Python>cd env_orders

D:\Dropbox\Python\env_orders>cd scripts

D:\Dropbox\Python\env_orders\Scripts>activate

(env_orders) D:\Dropbox\Python\env_orders\Scripts>
```

Note:

- If you want to remove a venv from your computer, call deactivate from the environment folder and then remove the folder through Explorer.

1.5.4 Install Django for the environment

```
..\env_orders>py -m pip install django==4.1.7
```

Check version:

```
..\env_orders>django-admin --version
```

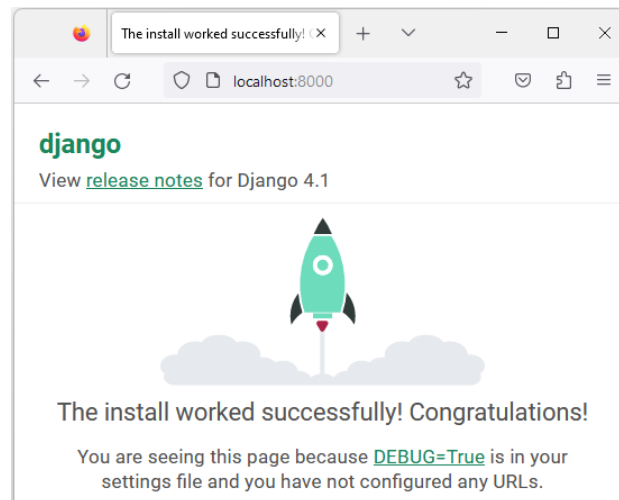
1.5.5 Create Django project inside the venv

```
..\env_orders>django-admin startproject pr_orders
```

1.5.6 Start server and check that it is running

In a separate command prompt:

- Activate: `..\env_orders\scripts>activate.bat`
- Start server: `..\env_orders\pr_orders>py manage.py runserver`
- You might see a message that there are some unapplied migrations. Ignore it. If you see a message “Quit the server with CTRL-BREAK”, it is fine.
- Check in browser: `localhost:8000`



- To stop server: Press Ctrl/Break (in the command window)
- Make sure to check the response in the command window regularly while debugging. There might be an error message there.

1.5.7 Prepare the editor

1.5.7.1 Install VS Code

Install Visual Studio code on your computer if it is not installed already. It can be downloaded from here: <https://code.visualstudio.com/download>.

1.5.7.2 Open the folder

Open VS Code

File / Open Folder ...

Select the folder that contains the `.venv` subfolder. In this case `env_orders`.

1.5.7.3 Check the extensions

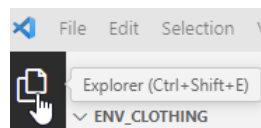
Check that the necessary extensions are installed:

- In VS Code, select View / Extensions (or type Ctrl+Shift+X).
- Check that at least the following extensions are installed:
 - Django (By Baptiste Darthenay)
 - Python (By Microsoft)
 - Pylance (Automatically added with Python)

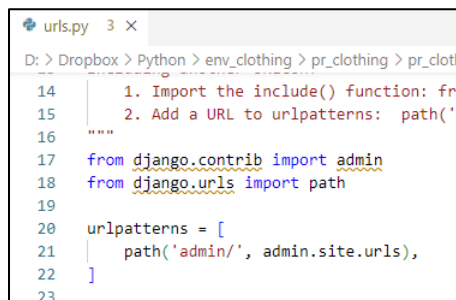
1.5.7.4 Check that the venv is recognised

Check that VS Code recognised the venv:

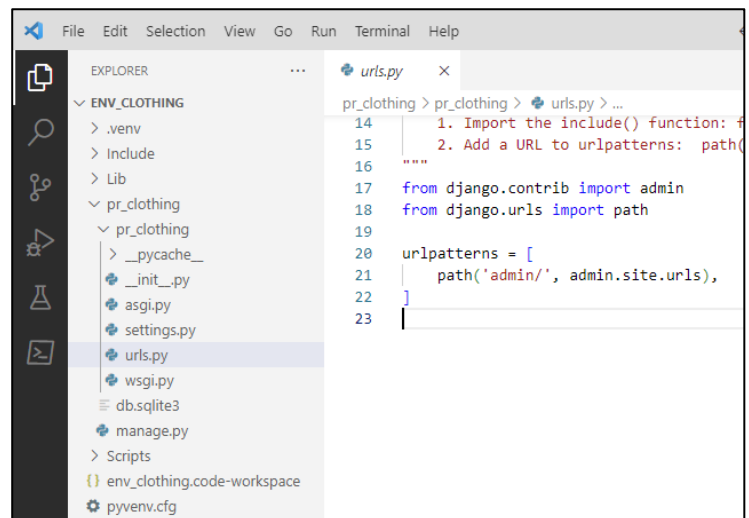
- Click on the Explorer tab in VS Code



- Open any Python file, e.g. urls.py. You should not get import-related errors (squiggly lines) in the .py files. There should be an indication in the bottom-right corner of the status bar that the venv orders is currently active.



This is wrong.

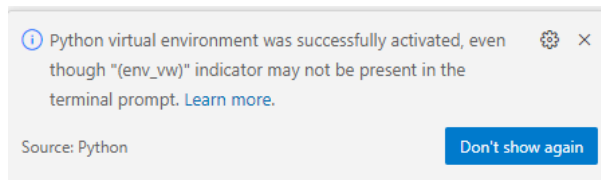


This is good.

- If you get errors, check the Python interpreter:
 - Ctrl-Shift-P
 - Python: Select Interpreter
 - Select the interpreter (python.exe) in the env_\scripts folder.

1.5.8 Create app

- In VS Code, open a PowerShell terminal (Terminal / New Terminal or Ctrl+Shift+`). (Of course, you can do this in the normal command window as well. The terminal in VS Code is just conveniently close now.)
- Activate the environment if it is not activated already. (If you activate from the VS Code PS terminal, you need to type `..\env_orders\scripts>.\activate`.)
- It is possible that VSCode will activate the environment automatically when you open a new terminal. Since the prompt does not indicate that the environment is activated, I prefer to not click on the 'Don't show again' button so that I will always know that the environment is activated.



- Browse to the project folder, `pr_orders`, and create the app inside the project folder:
`../pr_orders> python manage.py startapp app_orders`

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS D:\Dropbox\Python\env_orders\scripts> .\activate
(env_orders) PS D:\Dropbox\Python\env_orders\scripts> cd..
(env_orders) PS D:\Dropbox\Python\env_orders> cd pr_orders
(env_orders) PS D:\Dropbox\Python\env_orders\pr_orders> python manage.py startapp app_orders
(env_orders) PS D:\Dropbox\Python\env_orders\pr_orders> █
```

From <https://docs.djangoproject.com/en/4.1/intro/tutorial01/> :

What's the difference between a project and an app? An app is a web application that does something – e.g., a blog system, a database of public records or a small poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.

1.5.9 Configurations

1.5.9.1 wsgi.py

In the editor (VS Code), check `../pr_orders/pr_orders/wsgi.py`. It should be fine. You can remove the comment lines at the top if you want to.

```
import os
from django.core.wsgi import get_wsgi_application
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'pr_orders.settings')
application = get_wsgi_application()
```

- Note, if you copy and paste from here, make sure that you delete all extra leading spaces/tabs.
- Make sure to replace `pr_orders` with whatever the project name is.

1.5.9.2 settings.py (Changes from default)

- In the file, `../pr_orders/app_orders/apps.py`, check the exact name and case of the config class.
- In `../pr_orders/pr_orders/settings.py`:

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    'app_orders.apps.AppOrdersConfig' #Class name as in apps.py
]

ROOT_URLCONF = 'pr_orders.urls'
TIME_ZONE = 'Africa/Johannesburg'
ALLOWED_HOSTS = ['*']
```

- Make sure that you save the file after changes.

1.6 Development procedure

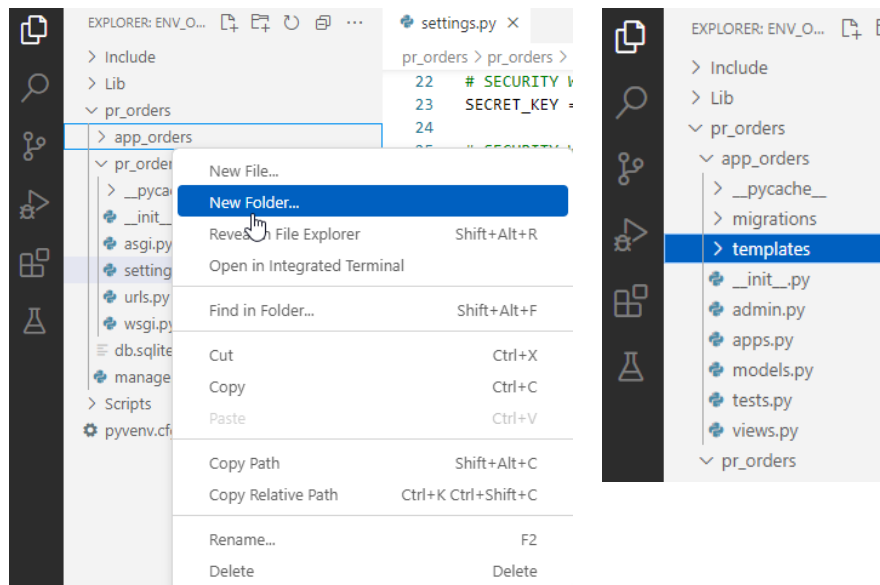
For every new page added to the website, we have to add three elements: A template, a view and a path. We will discuss these in more details later, but for now it suffices to say the following:

- All three elements are saved on the server.
- A template is an HTML file on which the rendered page will be based.
- A view is a Python function that will render the template and return it to the client to be displayed in a browser.
- A path is an entry in a Python array that maps the URL as entered by the client in a browser to the view name.

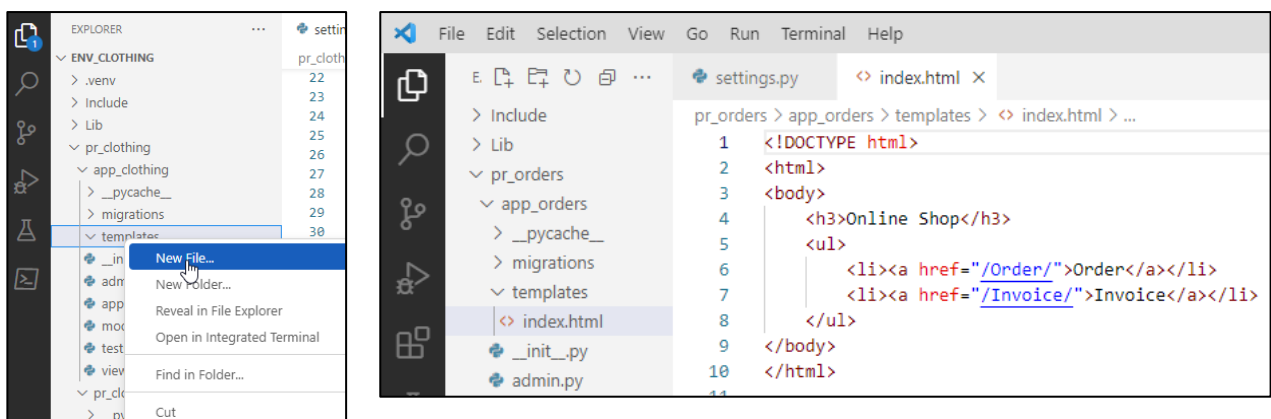
1.6.1 Template

The index is normally the starting point of a web application. From there, the user can browse to other pages of the site.

- In the app folder, create a folder, templates, if it does not exist already.




- In the templates folder, create a file, index.html (note lower case), with example content as below. Make sure that you save the file after changes.



- We will expand this project further in the following chapter. We will show how to activate the hyperlinks to show pages with a list of suppliers or customers or products.

1.6.2 View

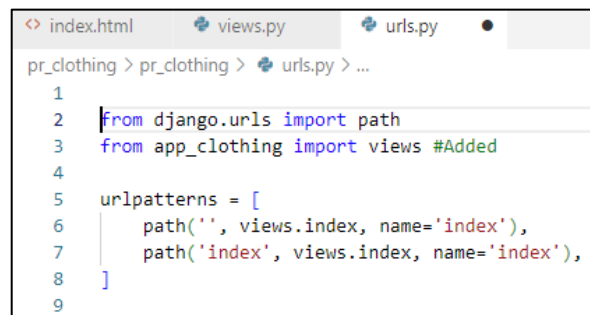
Enter the following text in the `views.py` file:



```
settings.py  index.html  views.py  X
pr_orders > app_orders > views.py > ...
1  from django.shortcuts import render
2
3  def index(request):
4      return render(request, 'index.html')
5
```

1.6.3 Paths

In the file `../pr_orders/pr_orders/urls.py` add the paths indicated below. Note the import of `views.py` in the `app_order` folder.



```
index.html  views.py  urls.py
pr_clothing > pr_clothing > urls.py > ...
1
2  from django.urls import path
3  from app_clothing import views #Added
4
5  urlpatterns = [
6      path('', views.index, name='index'),
7      path('index', views.index, name='index'),
8  ]
9
```

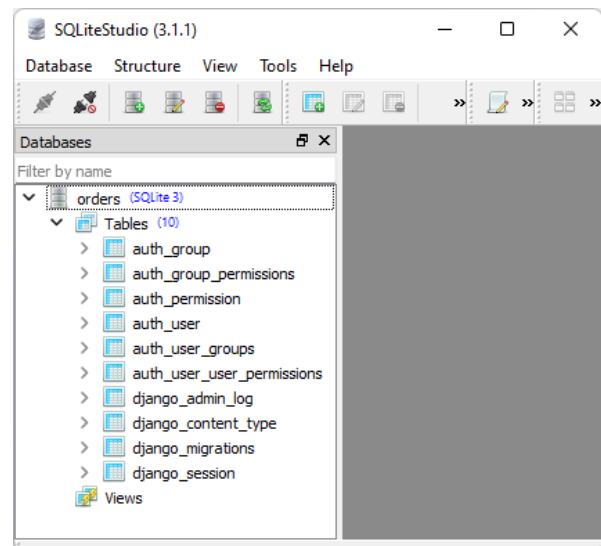
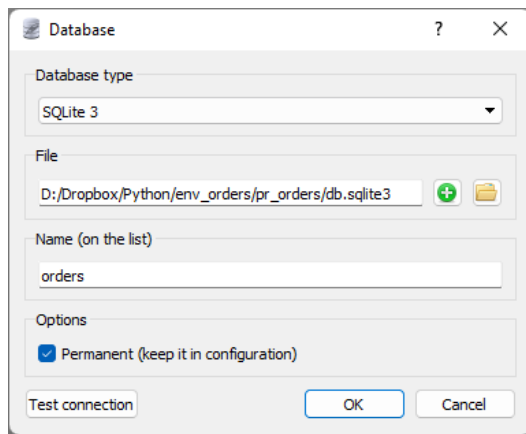
1.6.4 Migrate

By default, Django saves its data in a SQLite database. The database is saved in the file `db.sqlite3` in the outer project folder. You can use SQLite Studio or DBeaver to inspect the contents of the database.

Django uses several tables in a database for admin purposes. In order to create these tables, enter the following command in the console window:

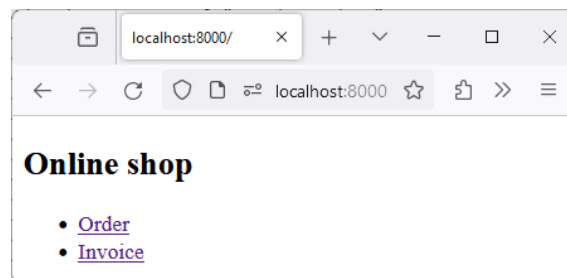
```
..\env_orders\pr_orders>python manage.py migrate
```

Use SQLite Studio and add the database. Then check that the admin tables were created.



1.6.5 Test

- Start server on localhost
`..\env_orders\pr_orders> py manage.py runserver`
- In a browser, if your paths are correct, these should all work:
 - <http://localhost:8000>
 - <http://localhost:8000/index>



- Notes:
 Of course, if you click on the links for Order and Invoice at this stage, you will get an error. We did not do that coding yet.
- Hint:
 Check the output of the command window where you started the server for possible error messages. Attend to the error, press Ctrl-Break to stop the server and start the server again.

1.6.6 Render immediate content

It is possible to render content directly from the view, i.e. without a template.

View

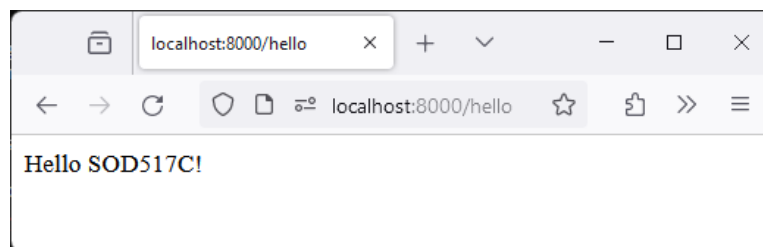
```
from django.http import HttpResponse
```

```
def hello(request):  
    return HttpResponse("Hello SOD517C!")
```

Paths

```
...  
urlpatterns = [  
    ...  
    path('hello', views.hello, name='hello'),  
]
```

Test

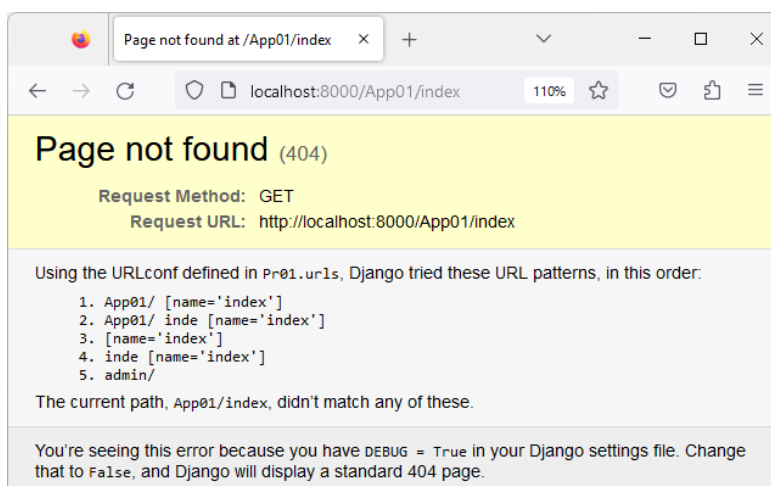


1.7 Debugging

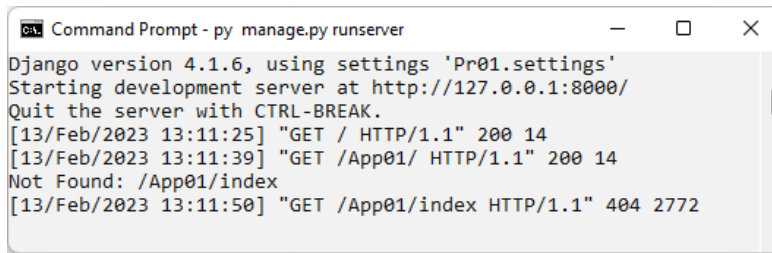
Feedback on a page is not always very helpful to identify errors. For example, if you spelled the route wrongly in `app.urls`:

```
urlpatterns = [  
    path('', views.index, name='index'),  
    path('Inde', views.index, name='index'),  
]
```

If the user types correctly localhost:8000/index, it will result in an 'HTTP 404 page not found' error.



Fortunately, an error message is displayed in the terminal window:



```
Command Prompt - py manage.py runserver
Django version 4.1.6, using settings 'Pr01.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
[13/Feb/2023 13:11:25] "GET / HTTP/1.1" 200 14
[13/Feb/2023 13:11:39] "GET /App01/ HTTP/1.1" 200 14
Not Found: /App01/index
[13/Feb/2023 13:11:50] "GET /App01/index HTTP/1.1" 404 2772
```

1.8 Summary

In this chapter, we introduced the development environment on localhost. We presented instructions to create a basic web app. The theory in the next chapter will refer to this example. It is essential that you get this right before we can proceed.

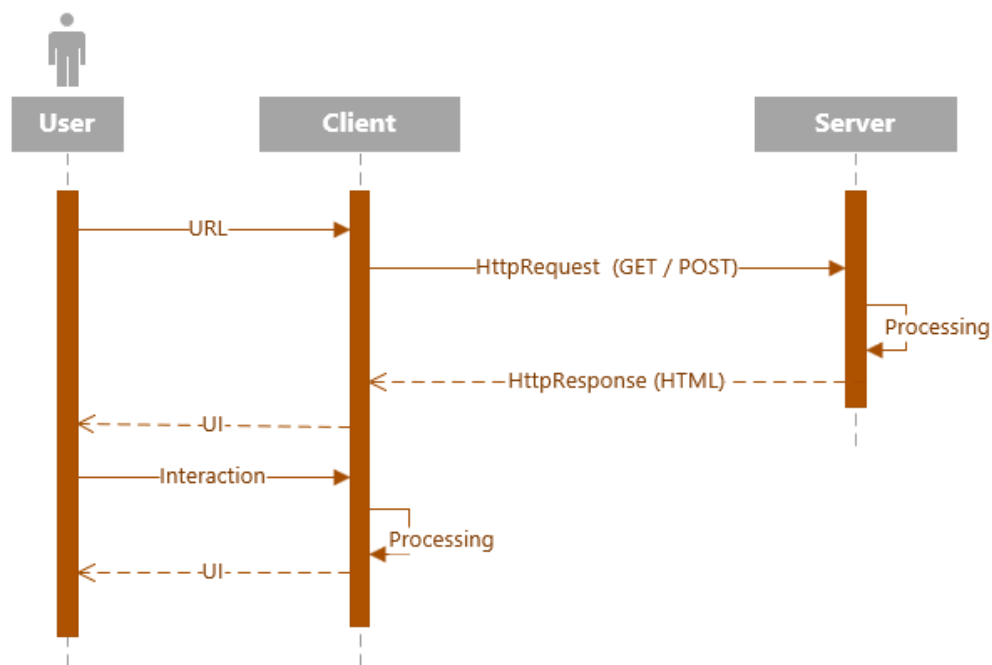
Chapter 2: Django basics

2.1 Server and Client

A browser runs on a user's PC, referred to as the client. It presents an interface with which the user can interact – referred to as the front-end or user interface (UI). The data serving the browser is located on a server, or the so-called backend.

Communication between the back end and front-end is over the internet according to the HTTP protocol. The client issues a request to the server which responds with a page that is displayed in the browser. Processing on the server is done in Python with the Django library installed. Processing on the client is in JavaScript.

This procedure is illustrated in the sequence diagram below.



Basic Client-Server setup

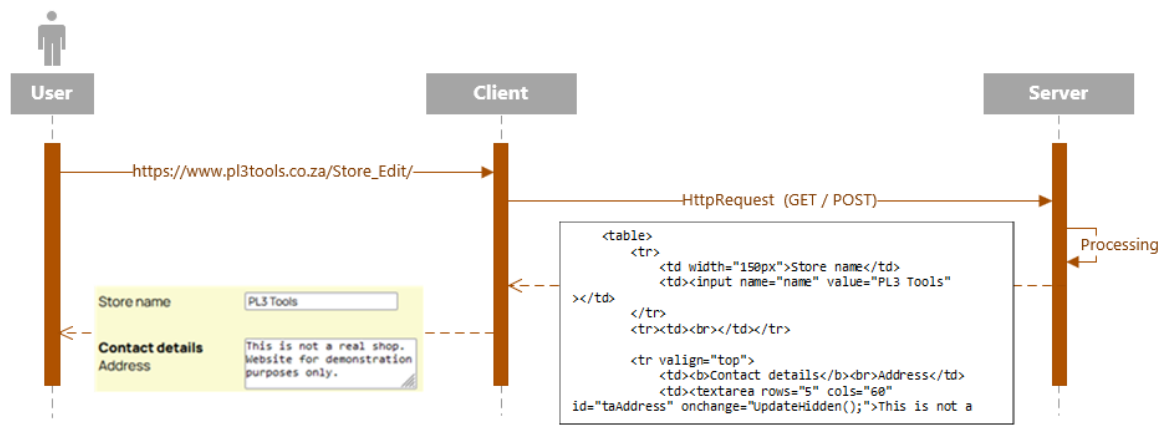
Further reading:

- <https://docs.djangoproject.com/en/5.0/ref/request-response/>

Sequence diagrams:

- A sequence diagram has actors and objects with a timeline that runs from top to bottom. Messages from one actor/object to another are indicated with arrows. Return messages are indicated with broken lines.
 - <https://www.lucidchart.com/pages/uml-sequence-diagram>
 - <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram>
 - <https://developer.ibm.com/articles/the-sequence-diagram>

Presentation on the client is done through HTML and CSS. If we replace the generics in the diagram above with something specific, it could look like this:



Example of specific request and response objects

2.2 What is Django?

Programming on the server is done in Python. Django is a Python library that takes care of much of the hassle of web development, so you can focus on writing your app without having to reinvent the wheel by doing everything in Python yourself. The drawback of this is that we have to fall in with the conventions and syntax of Django which may sometimes be a bit awkward.

Futher reading:

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

2.3 Design pattern

As opposed to the MVC pattern followed in ASP.Net, Django follows an MVT (model-view-template) design pattern. A Django view takes the role of a controller in ASP.Net. The Django template takes the role of the view in ASP.Net. The model in MVC and MVT fulfils more or less the same role.

	Element	Role	File extension
ASP.Net	Model	State of the application (data) and any business logic that should be performed by the model.	.cs
	View	Present content through the user interface (browser). (Template in MVC)	.cshtml
	Controller	Handles user interaction, work with the model and renders a view. (View in MVC)	.cs
Django	Model	The data that you want to present, usually from a database. (Model in MVC)	.py
	View	A Python function that takes http requests and return http responses, e.g. HTML documents. (MVC controller)	.py
	Template	A file that describes how the response should be presented in the client's browser. (MVC view)	.html

See also this:

<https://django.readthedocs.io/en/stable/faq/general.html#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names>
★ https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Home_page

2.4 Files and folders in Django

2.4.1 File structure

You can use either Windows File Explorer or the VS Code Explorer to inspect the file and folder structure of a web app. To distinguish between files and folders in the following schema, folders are followed by /. The schema below is not complete and there are several files and folders that are not shown.

```
env_orders/  
  .venv/  
  Include/  
  Lib/  
    site-packages/  
      django/  
      pip/  
      ... (Several others)  
pr_orders/ (Outer project folder)  
  manage.py  
  app_orders/  
    migrations/  
    templates/      (Created by you – see below)  
      base.html      (Created by you)  
      index.html  
      ... (Others created by you as needed for the project)  
  __init__.py  
  admin.py  
  apps.py  
  forms.py (Optional - see below)  
  models.py  
  tests.py  
  urls.py (Optional - see below)  
  views.py (Edited by you - often)  
pr_orders/ (Inner project folder)  
  __init__.py  
  asgi.py  
  settings.py  
  urls.py  
  wsgi.py  
  scripts/  
    activate.bat  
    deactivate.bat  
    django-admin.exe  
    pip.exe  
    python.exe  
    ... (Various others)
```

- The outer `Project/` root folder is a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.

- The inner `project/` folder is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. `mysite.urls`).

2.4.2 Reference to files

2.4.2.1 General notation

You should understand the way in which Django refers to folders, files, and modules. Instead of using `\` to indicate sub-folders, we use a period. We don't have to specify the entire path as Python searches the entire venv folder. Using this convention, we can also refer to modules inside a file.

So, `app_orders.apps.AppOrdersConfig` refers to the `AppOrdersConfig` class in the `apps.py` file in the `app_orders` folder.

In the folder notation, this would be

`D:\Dropbox\Python\env_orders\pr_orders\app_orders\apps.py`.

2.4.2.2 Importing

When we import existing modules, we follow the `from <folder>/<file>/<module> import <file>/<module>` syntax. Again, we don't have to specify the entire path - the Python interpreter will know to start searching from the environment folder.

```
from app_orders import views #..\env_orders\app_orders\views.py
```

Python will search all files in a subfolder for a specific module. So, these two lines are equivalent:

```
#..\env_orders\Lib\site-packages\django\conf.py\path
from django.urls.conf import path
from django.urls import path
```

If we import a module, we don't have to qualify it with its folder when using it. Compare the references to the `views` `index` and `hello` in the code below:

```
from django.urls import path #..\env_orders\Lib\site-packages\django\conf.py\path
from app_orders import views #..\env_orders\pr_orders\views.py
from app_orders.views import hello

urlpatterns = [
    path('', views.index, name='index'),
    path('index', views.index, name='index'),
    #No need to specify the folder below because the module has been imported
    path('hello', hello, name='hello'),
]
```

If we do not specify the `from` clause, Python searches for a module in the global Python installation. You might find this example in the `settings.py` file:

```
#C:\Users\<username>\Appdata\Local\Programs\Python\Python311\Lib\os.py
import os
```


2.4.3 Notes about some files and file types

All Django files are saved on the server in a fixed directory structure as expected by Django. Some customisations are possible by specifying them in the settings.py file.

2.4.3.1 Short notes

- `__init__.py`: An empty file that tells Python that this folder should be considered a Python package. Read [more about packages](#) in the official Python docs.
- `asgi.py`: An entry-point for ASGI-compatible web servers to serve your project. See [How to deploy with ASGI](#) for more details.
- `manage.py`: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about `manage.py` in [django-admin and manage.py](#).
- `models.py`: A model is a class, defined in Python, that provides an interface between some views and the database. Django takes care of the background stuff so that the programmer does not have to worry about database connections or issue SQL commands to query or update the database. The file `models.py` contain several such classes.
- `settings.py`: Settings/configuration for this Django project. [Django settings](#) will tell you all about how settings work.
- Template files: Templates are html files that describe how the response should be presented in the client's browser. They are saved on the server.
- `urls.py`: The URL declarations for this Django project; a “table of contents” of your Django-powered site. The file is used to map a URL as provided by the browser to a view. After a request, the server will start here to look for the requested path. Depending on the configuration, there may be more than one `urls.py` file – more about that later. You can read more about URLs in [URL dispatcher](#).
- `views.py`: A view is a Python function that takes an `HttpRequest` parameter and returns an `HttpResponse` object. The file `views.py` contain several such functions.
- `wsgi.py`: An entry-point for WSGI-compatible web servers to serve your project. See [How to deploy with WSGI](#) for more details.

Reference:

- <https://docs.djangoproject.com/en/4.1/intro/tutorial01/> :

2.4.3.2 `asgi.py` & `wsgi.py`

(Pronounced ass-ghee and whizz-ghee respectively.)

`wsgi.py` provides an entry-point for WSGI-compatible web servers to serve your project. `wsgi` is an acronym for web server gateway interface. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.

```
import os
from django.core.wsgi import get_wsgi_application
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'pr_orders.settings')
application = get_wsgi_application()
```

References

- [How to deploy with WSGI](#)
- <https://builtin.com/data-science/wsgi>
- <https://wsgi.readthedocs.io/en/latest/what.html>
- https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

- <https://www.fullstackpython.com/wsgi-servers.html>

asgi.py is the asynchronous version of wsgi.py. More information here:

- <https://docs.djangoproject.com/en/4.2/howto/deployment/asgi/>

2.4.3.3 manage.py

The manage.py file is inside the Project folder and imports Django's administrative commands.

- Make sure that the current folder is the project folder:
(env_orders) HH:mm ~\$ cd pr_orders.
- To list the available Django commands:
~/pr_orders \$ python manage.py
- To run a command, type python manage.py followed by the command that you want to execute and any parameters, for example:
~/pr_orders \$ python manage.py startapp app_orders
- If you want to execute a command to inspect a file that is situated in another folder, add the folder name. In the following example, the content of the 0001 file in the app_clothing folder will be displayed:
~/pr_orders \$ python manage.py migrate app_orders 0001

2.4.3.4 settings.py

This file contains all configuration settings for the Django installation. A typical settings file can look like this:

Short entries (in alphabetical order)

```
ALLOWED_HOSTS = ['*']
BASE_DIR = Path(__file__).resolve().parent.parent
DEBUG = True
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
LANGUAGE_CODE = 'en-us'
ROOT_URLCONF = 'pr_orders.urls'
STATIC_URL = 'static/'
TIME_ZONE = 'Africa/Harare'
USE_I18N = True
USE_TZ = True
WSGI_APPLICATION = 'pr_orders.wsgi.application'
```

Longer entries

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

INSTALLED_APPS = [
    ...
    'app_orders.apps.AppOrdersConfig'
]

MIDDLEWARE = [
    ...
]
```

```

TEMPLATES = [
    ...
]

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',
    },
]

```

Notes:

- `BASE_DIR = Path(__file__).resolve().parent.parent`
 - In Python, `__file__` refers to the absolute path of the current file, `settings.py`.
 - `settings.py` is inside the inner project folder which is inside the outer project folder. Thus, `parent.parent` refers to the outer project folder.
 - This means that, in my case, the `BASE_DIR` folder is `D:\Dropbox\Python\env_orders\pr_orders`.
- `ROOT_URLCONF = 'pr_orders.urls'`
 - Means that the mapping from URLs to views are saved in the file `urls.py` that resides in the sub-folder `pr_orders` of `BASE_DIR`.
- `INSTALLED_APPS`
 - Must contain an entry for the config file of your web app.
- `MIDDLEWARE`
 - Will be discussed later
- `TEMPLATES`
 - If you save the templates in the `app_.templates` folder, you don't have to do anything here. If you want save the templates elsewhere, you will have to indicate where they are. This will be discussed later.
- `DATABASES`
 - The default database of a new Django application is SQLite
 - The engine is in the folder `..\env_orders\Lib\django\db\backends\sqlite3`.
 - The database is in `BASE_DIR\db.sqlite3` which is `..\env_orders\pr_orders\db.sqlite3`.
- `AUTH_PASSWORD_VALIDATORS`
 - These rules are enforced for user administration. During debugging, we can comment them all out.

Further reading

- <https://docs.djangoproject.com/en/4.2/topics/settings/>

2.4.3.5 Templates

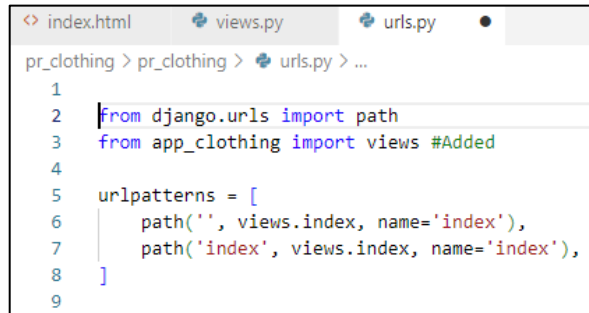
A template is an html document saved on the server that will determine the layout of the page that will be rendered to the client and displayed by the browser.

Further reading:

- <https://docs.djangoproject.com/en/4.1/#the-template-layer>
- <https://docs.djangoproject.com/en/4.1/ref/templates/>

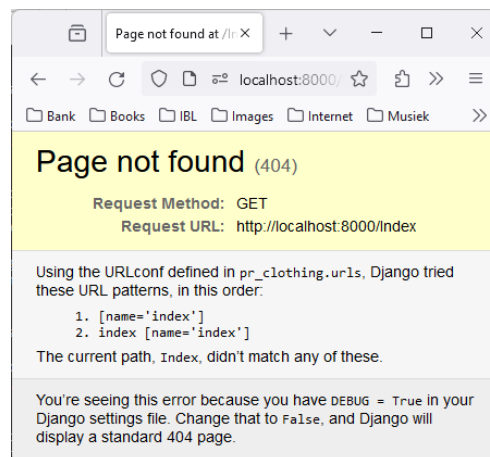
2.4.3.6 urls.py

The file is used to map a URL as provided by the browser to a view. After a request, the server will start here to look for the requested path. Depending on the configuration, there may be more than one urls.py file – more about that later. You can read more about URLs in [URL dispatcher](#).



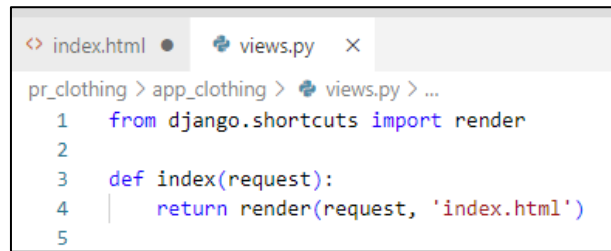
```
<> index.html  views.py  urls.py  ●
pr_clothing > pr_clothing > urls.py > ...
1
2 from django.urls import path
3 from app_clothing import views #Added
4
5 urlpatterns = [
6     path('', views.index, name='index'),
7     path('index', views.index, name='index'),
8 ]
9
```

- The entry for `ROOT_URLCONF` in the `settings.py` file defines what the browser url, <http://localhost:8000/>, will point to. In this case, `ROOT_URLCONF="pr_orders.urls"`, which means that the server will look in the file `../pr_orders/urls.py` for the requested path.
- The first path call means that if the user enters the root only, the view, `index`, will be called: <http://localhost:8000/>
- The second path call means that if the user enters the name of the page explicitly after the root, the view, `index`, will also be called: <http://localhost:8000/index>.
- Since Python is case sensitive, <http://localhost:8000/Index>, will result in an error. You can use regular expressions to resolve this issue.



2.4.3.7 views.py

A view is a Python function that takes an `HttpRequest` parameter and returns an `HttpResponse` object. The `HttpResponse` object contains the information that the browser will need to display a page. For small projects, all views can be contained inside `views.py`. For larger projects, create a folder, `views`, in the app folder.



```
<> index.html ● views.py ×
pr_clothing > app_clothing > views.py > ...
1  from django.shortcuts import render
2
3  def index(request):
4      return render(request, 'index.html')
5
```

- The first line is an import from a library.
- The view is a Python function. It starts with a def keyword and must return an HttpResponse object.
- In Python, indenting indicates the beginning and end of code blocks. So, the indenting is crucial here. The view starts with Line 3 and ends with Line 4.
- Make sure that you save the file after changes.

Further reading:

- <https://docs.djangoproject.com/en/4.1/#the-view-layer>

2.4.4 Static files

Static files such as js, css, and image files should be placed under a dedicated static folder under app.

- Create folders app_/static/js, app_/static/css and app_/static/img folders to save the different types of files.
- In settings.py:

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    ...
]
```

```
STATIC_URL = 'static/'
```

- In the templates:

```
<head>
    {% load static %}
</head>
```
- To display an image:

```

```
- To include a js file:

```
<head>
    {% load static %}
    <script src="{% static '/js/script.js' %}"
        type="text/javascript"></script>
</head>
```
- To include a css file:

```
<head>
```

```

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

```

Further reading:

- <https://www.javatpoint.com/django-static-files-handling>

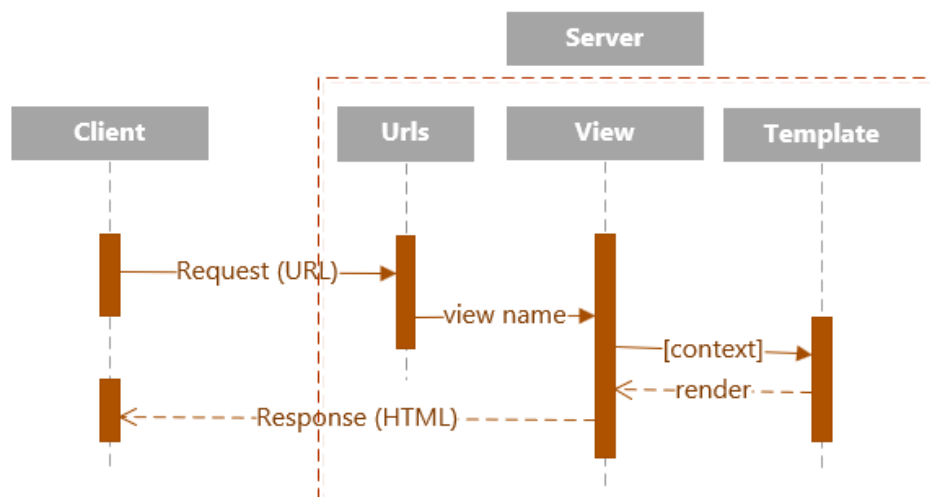
2.5 HTTP is stateless

HTTP is a stateless protocol (https://en.wikipedia.org/wiki/Stateless_protocol). This means that once the client sent a request to the server, it cannot update the request. Also, once a server responded, it forgets about the request and the web page is independent. There is no shared storage between the client and the server. If the user wants to change something small, for example the details of a filter of data that is requested, he has to submit the request again and the server must start all over again to fetch the data and respond with a new page that contains the new data.

2.6 Sequence of events

The sequence diagram below shows an example of the process that is followed when a client requests the index page of a web app.

1. The client issues a GET request to the server.
2. The urls.py file maps the given URL to a view name with parameters. Remember that views are functions.
3. The view retrieves the applicable template. If needed, it executes the embedded code in the template and fills the placeholders (`{{ }}` tags) with data (referred to as the context). ([context] is written in square brackets below because it is optional.)
4. The view returns an `HttpResponse` object to the client. This object contains the page source that the browser will use to display the page.



Sequence of events when requesting a basic web page

2.7 Resources

Web development

- <https://intellipaat.com/blog/python-web-development-tutorial/>
- <https://realpython.com/tutorials/web-dev/>
- <https://realpython.com/learning-paths/become-python-web-developer/>
- <https://www.educative.io/blog/web-development-in-python>
- <https://www.fullstackpython.com/introduction.html>
- <https://www.fullstackpython.com/web-development.html>
- https://www.tutorialspoint.com/python_web_development_libraries/index.htm
- https://www.tutorialspoint.com/python_web_development_libraries/index.htm
- https://www.tutorialspoint.com/python_web_development_libraries/python_web_development_libraries_introduction.htm (Remove space)

Flask

- <https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3>
- <https://www.educative.io/blog/web-development-in-python>
- <https://www.javatpoint.com/flask-tutorial>
- <https://www.softwaretestinghelp.com/python-flask-tutorial/>
- <https://www.tutorialspoint.com/flask/index.htm>
- https://www.tutorialspoint.com/python_web_development_libraries/python_web_development_libraries_flask_framework.htm (Remove space)

Django

- <https://docs.djangoproject.com/en/4.1/intro/tutorial01/>
- <https://medium.com/javarevisited/my-favorite-courses-to-learn-django-for-beginners-2020-ac172e2ab920>
- <https://www.tutorialspoint.com/django/index.htm>
- https://www.tutorialspoint.com/python_web_development_libraries/python_web_development_libraries_django_framework.htm (Remove space)
- <https://www.w3schools.com/django/>
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/skeleton_website
- <https://devdocs.io/django~4.1/intro/install>

Flask vs Django

- <https://hackr.io/blog/flask-vs-django>
- <https://intellipaat.com/blog/python-web-development-tutorial/> (Scroll down)
- <https://pythonprogramming.net/web-development-tutorials/>
- <https://www.geeksforgeeks.org/differences-between-django-vs-flask/>
- <https://www.guru99.com/flask-vs-django.html>
- <https://www.guru99.com/flask-vs-django.html>
- <https://www.simplilearn.com/flask-vs-django-article>

For this course, we will follow the Django framework.

2.8 Summary

In this chapter we discussed some of the basic Django concepts based on the example web app that was developed in Chapter 1. We referred to the work that is done on the server and client respectively, the design pattern followed by Django as well as the file and folder structure. We also provided some short notes on some the files and file types.

Chapter 3: Front-end development

3.1 Introduction

It is assumed that you are well-versed and familiar with client-side web development using html, css and JavaScript. If not, see this: <https://developer.mozilla.org/en-US/docs/Learn>.

Some links that you may use to refresh your knowledge in your own time are given in Appendix A.

3.2 HTML forms

Note: Please don't confuse the concepts of HTML forms with Django forms. This section is about HTML forms. Django forms are classes that can take the burden of developing HTML templates away from the developer. To my mind, this gives me less control over the visual layout of a page, and I only use them in rare cases.

In the examples below we show how to get order data from a user and then display an invoice.

Order product	Invoice
<p>Product name <input type="text" value="Ryobi 115 mm grinder"/></p> <p>Amount <input type="text" value="780"/></p> <p style="text-align: center;"><input type="button" value="Place order"/> <input type="button" value="Cancel"/></p>	<p>Product name: Ryobi 115 mm grinder</p> <p>Amount: 780</p> <p style="text-align: center;"><input type="button" value="Index"/></p>

3.2.1 Paths

We will have two views for the order and therefore we need two paths – one for GET and one for POST. The user might enter the url for GET directly in a browser, but they will never need to enter the url for POST explicitly.

```
urlpatterns = [
    ...
    path('order/', order),
    path('order_post/', order_post),
    path('invoice/', invoice),
]
```

Notes:

- It is always important to check that the naming of views and templates are consistent across all references thereto. A common mistake is that we are inconsistent with the usage of lower case/upper case. The golden rule is to always use all lower case. Separate words with an underscore. See <https://realpython.com/python-pep8/#naming-conventions> again.
- The urls are case sensitive and end users might complain that your links do not work. You can solve the problem by using regular expressions and use `re_path` instead of `path`:


```

from django.urls import re_path
urlpatterns = [
    ...
    re_path(r'(?i)order/', order),
    re_path(r'(?i)order_post/', order_post),
    re_path(r'(?i)invoice/', views.invoice),
]

```

3.2.2 Templates

3.2.2.1 Order

When it is necessary to enter data and post back to the database, we need an HTML form with method="post".

```

<form action="/order_post/" method="post">
    {% csrf_token %}
    <table>
        <tr>
            <td colspan="2" align="center">
                <h1>Order product</h1>
            </td>
        </tr>
        <tr>
            <td><b>Product name</b></td>
            <td><input name="product_name" required></td>
        </tr>
        <tr>
            <td><b>Amount</b></td>
            <td><input type="number", min="0.00" step="any" name="amount" required></td>
        </tr>
        <tr>
            <td><br></td>
        </tr>
        <tr>
            <td colspan="2" align="center">
                <input type="submit" value="Place order">
                <input type="button" value="Cancel" onclick="window.location='/';">
            </td>
        </tr>
    </table>
</form>

```

Notes

- The action indicates what should happen when the user clicks the Submit button. In this case, the urls.py entry will call the order_post method.
- The tag, {% csrf_token %}, is a Django tag to prevent cross site request forgery (<https://docs.djangoproject.com/en/4.1/ref/csrf/>). This is an essential tag for Django and you will get an error if you omit it. Inspect the page source in the browser to see how this tag is interpreted.
- NB: The name attribute for input tags will allow the view to read the values after post.
- Note the difference in the type of the input tags for two buttons.
- The submit button will close the form and redirect to the URL specified in the action attribute of the form. This will, in turn, call the order_post view.
- The cancel button redirects to the home page, which is index.

Further reading:

- <https://docs.djangoproject.com/en/4.2/topics/forms/>

3.2.2.2 Invoice

```
<table>
  <tr>
    <td colspan="2" align="center">
      <h3>Invoice</h3>
    </td>
  </tr>
  <tr>
    <td><b>Product name:</b></td>
    <td>{{ product_name }}</td>
  </tr>
  <tr>
    <td><b>Amount:</b></td>
    <td>{{ amount }}</td>
  </tr>
  <tr>
    <td colspan="2" align="center">
      <br>
      <input type="button" value="Index" onclick="window.location = '/';">
    </td>
  </tr>
</table>
```

Notes

- We don't need a <form> tag here – we are just displaying data and nothing will be submitted.
- The data is displayed where the Django {{ }} tags are.

3.2.3 The views

3.2.3.1 Order

We need two views – one to render the form and one to submit the data that the user entered on the form. It is possible to do this in one view, but I believe it is less confusing to do it in two separate views.

```
from django.shortcuts import render
from . import vw_home

def order_get(request):
    return render(request, 'order.html')

def order_post(request):
    if request.method == "POST":
        product_name = request.POST['product_name']
        request.session['product_name'] = product_name
        amount = request.POST['amount']
        request.session['amount'] = amount
        #return render(request, 'index.html')
        return vw_home.index(request)
```

Notes:

- The first view just renders the order.html page.
- The second view is called after the user submitted the form.
- First, we need to check if the form was submitted with method=post.
- Then we can retrieve the data that the user entered with the request.POST dictionary as indicated in the example.

<https://docs.djangoproject.com/en/5.0/ref/request-response/#django.http.HttpRequest.POST>

- Normally, we would use the retrieved data to update the database. Since we did not discuss the usage of a database yet, we just copy the data into session variables for the time being.
- After processing, we return the user to the index page – either by calling the index view or rendering the index page directly.

3.2.3.2 Invoice

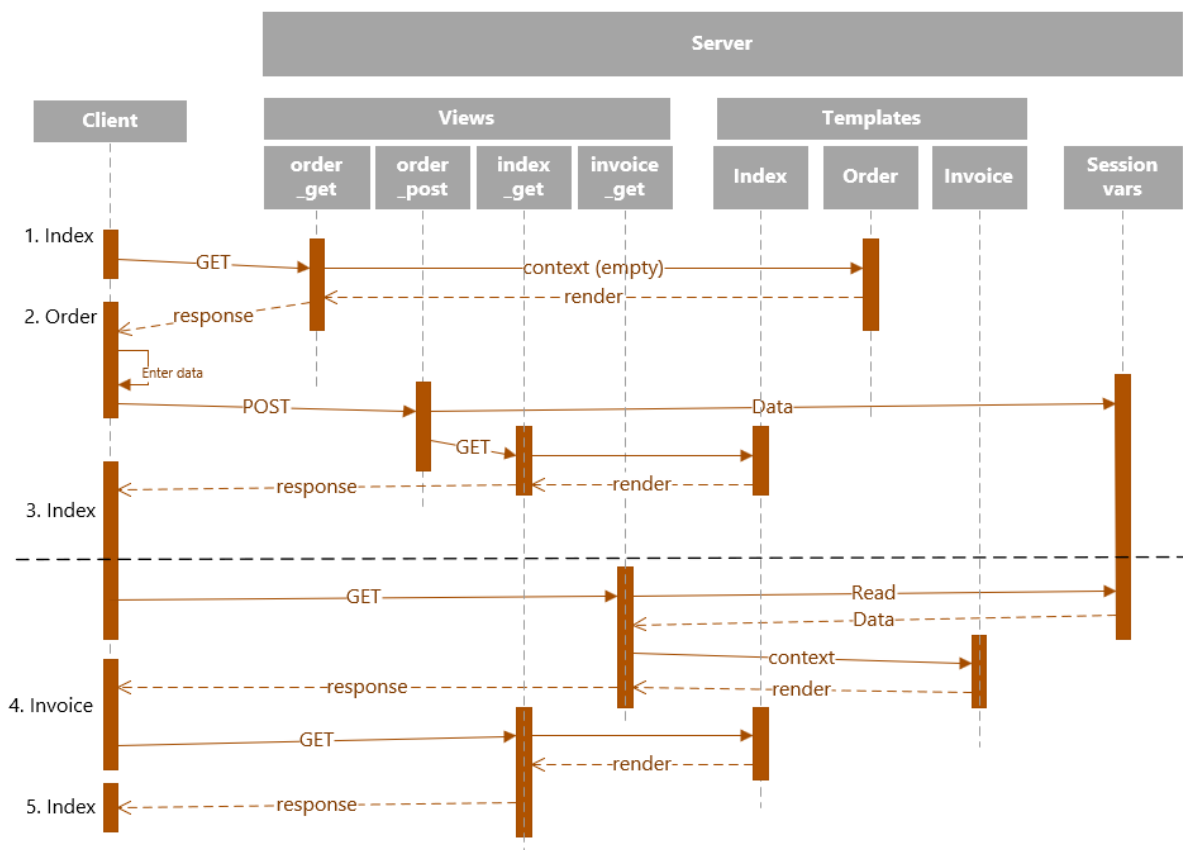
```
def invoice_get(request):
    product_name = request.session['product_name']
    amount = request.session['amount']
    return render(request, "invoice.html", \
        {'product_name': product_name, 'amount': amount })
```

Notes:

- For now, we acquire the data from the session variables that were created for the order. Normally, this would be retrieved from the database.
- The data is sent through to the template via the context parameter.

3.2.4 Sequence diagram

The sequence diagram below shows order of events for the above procedure. To avoid clutter a bit, the paths interface has been omitted.



Order of events for order and invoice

At a first glance, the diagram might seem a bit overwhelming. Remember that a sequence diagram has timelines, and the content must be read from left to right and top to bottom. Let us break it up and walk through it step-by-step.

There are two separate processes as indicated by the horizontal broken line: One for ordering a product and one for viewing the invoice. Each process starts and ends with the index page. The session variables span both processes.

1. The procedure starts with index page.
 - 1.1 The client submits a request to order a product.
 - 1.2 The `order_get` view acquires the applicable template, renders it, and returns an `HttpResponse` object (a blank order page) that will be displayed on the client.
2. The blank order page is displayed on the client.
 - 2.1 The client enters some data and submit the form.
 - 2.2 The `order_post` view saves the data as session variables.
 - 2.3 The `order_post` view calls the `index_get` view. The index template is acquired, and the page is rendered.
3. The index is displayed again
 - 3.1 The client requests to view the invoice.
 - 3.2 The `invoice_get` view reads the data from the session variable
 - 3.3 The `invoice_get` view sends the data through the context parameter to the template, renders the filled invoice page and returns it to the client.
4. The invoice is displayed
 - 4.1 The client views the invoice and requests to return to the index.
 - 4.2 The `index_get` view acquires the index template and renders it.
5. The index page is displayed again.

3.3 Using templates

3.3.1 Server and client versions of the same html file

Don't confuse the template, which is an html document saved on the server, with the html document that lives on the client. The template is what its name says – it's a template, i.e. the basis from which a rendered html page is built.

Compare the extracts from a template and its rendered version line-by-line in the example below. The template contains place holders for actual data between `{{ }}` and programming constructs between `{% %}`. The rendered document contains actual data and executed versions of the programming constructs in the template.

Template (on server) invoice.htm	Rendered html (on client) invoice.html
<pre> <table> <tr> <td colspan="2" align="center"> <h2>Invoice</h2> </td> </tr> <tr> <td>Product name:</td> <td>{{ product_name }}</td> </tr> <tr> <td>Amount:</td> <td>{{ amount }}</td> </tr> <tr> <td colspan="2" align="center">
 <input type="button" value="Index" onclick="window.location = '/';"> </td> </tr> </table> </pre>	<pre> <table> <tr> <td colspan="2" align="center"> <h2>Invoice</h2> </td> </tr> <tr> <td>Product name:</td> <td>Table saw</td> </tr> <tr> <td>Amount:</td> <td>8911</td> </tr> <tr> <td colspan="2" align="center">
 <input type="button" value="Index" onclick="window.location = '/';"> </td> </tr> </table> </pre>

The above page is displayed in the browser like this:

Invoice

Product name: Table saw

Amount: 8911

3.3.2 Django tags

Django has special tags to allow embedding in html as shown in the template example above. Below are some more listed.

- Comments


```
{#   #}
```
- Comment block


```
{% comment $}
    Something else
{% endcomment %}
```
- Template inheritance


```
{% extends "<parent>.html" %}
{% block <blockname> %}
    Some html
{% endblock %}
```

 - The content of the named block in the parent html will be replaced with the content of the same named block in child html files.
 - See <https://docs.djangoproject.com/en/4.0/ref/templates/language/#template-inheritance>
- Variables. The value of a variable is printed instead of:


```
{{ variable }}
```

- Conditions


```
{% if condition %}
    Some html
{% else %}
    Some html
{% endif %}
```
- Loops


```
{% for item in item_list %}
    Some html
{% endfor %}
```
- Cross site request forgery protection


```
{% csrf_token %}
```

See <https://docs.djangoproject.com/en/4.1/ref/csrf/> for details.
- Load a custom template set


```
{% load %}
```

Example: Load all the files (styles, images, etc.) in the static folder.

```
{% load static %}
```

Beware: If there are too many large files in the static folder, it will slow down the initial load of the page.

See <https://docs.djangoproject.com/en/4.2/ref/templates/builtins/#load> for details.

Notes

- There are no closing : as in Python.
- It is not advisable to do complex logic in a template. Do logic as far as possible in the views.

References

- Variables
 - https://www.w3schools.com/django/django_template_variables.php
- Tags
 - https://www.w3schools.com/django/django_template_tags.php
 - <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>
- Control structures
 - https://www.w3schools.com/django/django_tags_if.php
 - https://www.w3schools.com/django/django_tags_for.php
- Comments
 - https://www.w3schools.com/django/django_tags_comment.php

3.3.3 Templates in subfolders

For large projects, it is advisable to save related templates in subfolders under the templates folder. Update the `TEMPLATES` entry in `settings.py` so that Django will know where to find the templates:

```
TEMPLATES = [
    {
        'DIRS': ['app_orders/templates',
                  'app_orders/templates/home',
                  ...
                ],
        'APP_DIRS': True,
        ...
    },
    ...
]
```

```
    },  
]
```

3.3.4 Formatting output

Output numbers in template

```
{{ product.net_price |floatformat:2}}
```

Preserve line breaks

```
{{ message | linebreaksbr }}
```

Further reading:

- https://www.w3schools.com/django/django_ref_filter.php

3.3.5 The context parameter

The render function in the view can optionally specify a context parameter. The context parameter facilitates communication from the view to the template. The context is a Python dictionary object with keys and values

(https://www.w3schools.com/python/python_dictionaries.asp). Python dictionaries are written between { }, for example {'key1': value1, 'key2': value2, 'key3': value3}.

The template will read the context and display the value of a specific entry in the dictionary where the key is given between {{ }} tags. It is essential that the context keys are spelled identically in the template and the view.

3.3.6 Example

View

The render function returns an HttpResponse object. Note how the context is specified.

```
def invoice_get(request):  
    product_name = request.session['product_name']  
    amount = request.session['amount']  
    return render(request, "invoice (without base).html", \  
                  {'product_name': product_name, 'amount': amount })
```

Template

When rendering the response, the return function of the view uses the context items to fill up the applicable placeholders in the template.

```
<tr>  
    <td><b>Product name:</b></td>  
    <td>{{ product_name }}</td>  
</tr>  
<tr>  
    <td><b>Amount:</b></td>  
    <td>{{amount|floatformat:2}}</td>  
</tr>
```

3.4 Views

A view is a Python function. All views are functions, but not all functions are views. To be regarded as a view, the function must (i) accept a HttpRequest object as a parameter and (ii) return a HttpResponse object.

- <https://docs.djangoproject.com/en/5.0/topics/http/views/>

3.4.1 Request and Response

An `HttpRequest` object has attributes such as `scheme`, `body`, `path`, `path_info`, etc. that can be used in a view to identify aspects of the request, for example the original url (`path` and `path_info`) that was issued by the client. `HttpRequest` has a few methods of which `GET` and `POST` are the most important. `GET` is used to request information from the server while `POST` is used to submit information to the server and make a change to the underlying database. In both cases, communication is from client to server.

An `HttpResponse` object contains the information that the browser will need to display a page. `HttpResponse` objects can be of various subclasses to provide for HTML documents, files, videos, etc. Communication is from server to client.

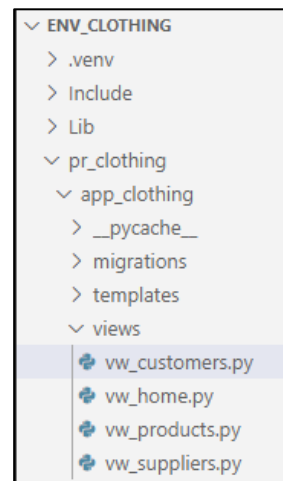
The attributes of `HttpResponse` include `content`, `charset`, `status_code`, etc.

Further reading:

- <https://docs.djangoproject.com/en/5.0/ref/request-response/>
- <https://www.javatpoint.com/django-request-and-response>
- <https://www.geeksforgeeks.org/django-request-and-response-cycle-httprequest-and-httpresponse-objects/>

3.4.2 Views in separate files

For a large project, it is advisable to save related views in a separate file. Create a folder, `views`, that contain separate views files. The file names should be informative and indicative of their role in the applications. See the screen print for an example.



3.5 Messages

Before we go any further it is essential to help ourselves with debugging. When we do desktop programming, we are used to add breakpoints to our code to assist us to follow the logic of a program and determine where things are going wrong. With web programming, the development environment and the running server are disconnected and breakpoints are not possible.

This section will discuss a technique to insert temporary message generating commands so that we can inspect the values of variables at various stages of program execution. We will discuss two possible alternatives, both of which are very useful.

3.5.1 A simple message box

In this example, we will show you how to display a simple message box.

3.5.1.1 Template

Under the templates folder, create a file, message.html:

```
<body>
  <h2>MESSAGE</h2>
  <pre>{{ message }}</pre>
  <div align="center">
    <input type="button" value="OK" text-align="center" onclick="window.location='/'">
  </div>
</body>
```

Notes:

- The message itself is displayed between `<pre>` `</pre>` tag markers. Text in a `<pre>` element is displayed in a fixed-width font, and the text preserves both spaces and line breaks. The text will be displayed exactly as written in the HTML source code.
- The message is sent through from the view via the context parameter and displayed in the Django tag `{{ }}`.
- The code is not complete. Only the essence is shown here. See the online documentation for the complete code.

3.5.1.2 View

```
def show_message(request, message):
    return render(request, 'message.html', {"message": message } )
```

3.5.1.3 Path

Unless you want the user to be able to display messages or access the messages from a template, we do not have to provide a path. We will call the message box only from within the views.

3.5.1.4 Usage

```
def something
... #Do something
return show_message(request, "Your message goes here")
```

For example, you can change the `order_post` view in the example above to show a message box when the product name is retrieved from the form.

```
def order_post(request):
    if request.method == "POST":
        product_name = request.POST['product_name']
        return vw_home.show_message(request, product_name)
        request.session['product_name'] = product_name
        amount = request.POST['amount']
        request.session['amount'] = amount
    return vw_home.index(request)
```

Notes:

- The view must return an `HttpResponse` object - therefore we must call `show_message` with `return`.

- The OK button in the message template redirects to the index, so we will go back to the index irrespective of the return path of the view that called the message box.
- The code following the message box will be greyed out since it is unreachable. We use a message box only for debugging and the call must be removed or commented out as soon as the issue has been resolved.

3.5.2 The messages framework

Django provides a framework with which we can accumulate messages over time and then flush the message storage once we are ready. This framework uses a messages class that contains messages in a list. The list is cleared when the messages are retrieved.

Further reading:

- <https://docs.djangoproject.com/en/4.2/ref/contrib/messages/>
- <https://docs.djangoproject.com/en/4.1/ref/contrib/messages/#adding-a-message>

3.5.2.1 Accumulate the messages

We can add messages to the storage in a view like this. Note the import.

```
from django.contrib import messages

def order_post(request):
    if request.method == "POST":
        product_name = request.POST['product_name']
        messages.info(request, "Product name: " + product_name)
        request.session['product_name'] = product_name
        amount = request.POST['amount']
        messages.info(request, "Amount: " + amount)
        request.session['amount'] = amount
    return vw_home.index(request)
```

Other types of messages are also possible:

```
messages.debug(request, "...")
messages.error(request, "Error: Authentication failed.")
messages.info(request, "...")
messages.success(request, "...")
messages.warning(request, "...")
```

3.5.2.2 Display the messages

Add this code to the bottom of all templates.

```
{% if messages %}
    {% for message in messages %}
        {{ message }}
        </br>
    {% endfor %}
{% endif %}
```

For example, add the code above to the index page. When the user enters the product details when ordering, the product name and price will be displayed at the bottom of the index page as soon as the user submits the form.

Order product

Product name

Amount

Online shop

- [Order](#)
- [Invoice](#)

Messages

Product name: Ryobi 115 mm grinder
Amount: 780

3.6 Template inheritance

Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines **blocks** that child templates can override. This means that we do not have to add similar content to every template.

Ref: <https://docs.djangoproject.com/en/4.0/ref/templates/language/#template-inheritance>

3.6.1 Unnecessary repetition

Until now, although not shown in this document, all templates included some styles and layout details that were more or less similar. The complete `index.html` file is given below.

```
<!DOCTYPE html>
<html>

<head>
  <style>
    body { text-align: center; }
    .div_center { display: inline-block; border: 1px solid; }
    ul, li, table { text-align: left; }
    ul { padding-left: 20px; }
    h2 { padding: 0px 10px 0px 10px; }
  </style>
</head>
<body>
  <div class="div_center">
    <h2>Online shop</h2>
    <ul>
      <li><a href="/order/">Order</a></li>
      <li><a href="/invoice/">Invoice</a></li>
    </ul>

    <!-- Messages -->
    <div style="text-align: left;">
      {% if messages %}
        <h3>Messages</h3>
        {% for message in messages %}
          {{ message }}
        </br>
        {% endfor %}
      {% endif %}
    </div>
  </div>
</body>
</html>
```

3.6.2 base.html

Under templates, create a new file, base.html, and enter following code.

```
<!DOCTYPE html>
<html>

<head>
  <style>
    body { text-align: center;}
    .div_center { display:inline-block; border: 1px solid;}
    ul, li, table { text-align: left; }
    ul { padding-left:20px; }
    h2 { padding:0px 10px 0px 10px}
  </style>

  <!--In the templates, surround the content also with <script></script> tags in the
    templates. If we put the tags here, we would lose the JavaScript intellisense in
    the templates. -->
  {% block scripts %}
  {% endblock scripts %}
</head>

<body>
  <div class="div_center">
    {% block content %}
    {% endblock content %}

    <!-- Messages -->
    <div style="text-align: left;">
      {% if messages %}
      <h3>Messages</h3>
      {% for message in messages %}
      {{ message }}
      </br>
      {% endfor %}
      {% endif %}
    </div>
  </div>
</body>
</html>
```

3.6.3 Inheriting from base.html

Now we can replace the content of index.html with the following:

```
{% extends "base.html" %}
{% block content %}
  <h2>Online shop</h2>
  <ul>
    <li><a href="/order/">Order</a></li>
    <li><a href="/invoice/">Invoice</a></li>
  </ul>
{% endblock %}
```

Notes:

- The {% extends %} tag indicates that the rendered page will include all content of base.html and extend it with the extra content in index.html.
- The {% extends %} tag must be the very first line in the file.
- The block with a specific name in base.html will be replaced with the block with the same name in index.html.
- In the browser, inspect the page source to check that the files index.html and base.html were integrated correctly.

3.7 Provide for mobile device (Enrichment)

- In command window or terminal:
`..\env_orders>py -m pip install django_user_agents`
- Create separate templates with the elements arranged such that they will fit on a mobile device. Save with suffix '_mobile'.
- In the views:

```
from django_user_agents.utils import get_user_agent

def index (request):
    user_agent = get_user_agent(request)
    if user_agent.is_pc: html = "index.html"
    else: html = "index_mobile.html"
    return render_page (request, html)
```

3.8 Summary

This chapter provided some links to assist you with revision of the tools that you need for front-end development. We also highlighted some basic html concepts and showed how to centre content in a browser window. We showed the details of a simple message box in Django.

Chapter 4: Data storage and access

4.1 Introduction

For interactive, data-driven websites, we need to save data in a permanent way. This is done in a database on the server.

4.2 Cookies and Session variables

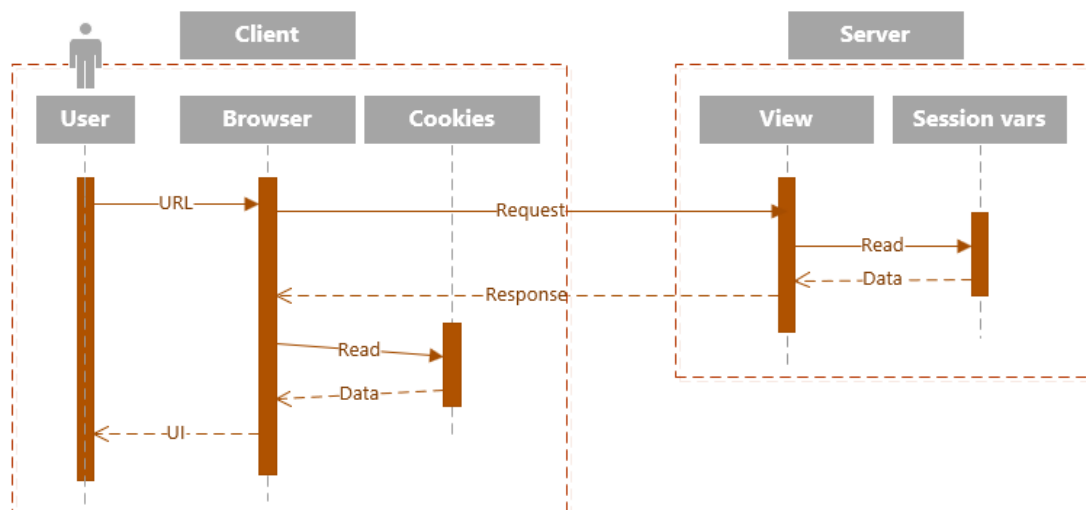
The bulk of data is saved in the database, but there is also a possibility of cookies on the client side and session variables on server side. These storage units reside on a specific platform and cannot be shared across server/client boundaries unless it is transmitted through a Request or Response object. Cookies can be permanently saved on the client, but session variables expire when the client closes the browser.

Cookies are saved on disk and in memory on the client PC and can be used to maintain state from one page to another in the browser. Session variables are saved on the server and can be used to maintain state from one request to another within the same session. A session is opened when a client request is received and closed when the browser on the client PC is closed.

The client cannot read session variables from the server and the server cannot access cookies on the client. Client and server are two independent entities and once a request is issued and a response is delivered, each lives in its own world.

4.2.1 Sequence diagram

The sequence diagram below shows the interaction with cookies and session variables.



Interactions with cookies and session variables

4.2.2 Cookies

We can save a value in a cookie on one page and then access it in another on the same site and during the same browser session.

4.2.2.1 Save cookie

A cookie is a key-value pair that can be set by assigning to the `document.cookie` object. Options can be set and separated by semi-colons.

```
<!DOCTYPE html>
<head>
  <script>
    function PlaceOrder()
    {
      amount = document.getElementById('id_amount').value;
      document.cookie = "amount=" + amount + ";path="/;
    }
  </script>
</head>

<html>
<body>
  <input type="button" value="Place order" onclick="PlaceOrder();">
</body>
</html>
```

4.2.2.2 Read cookie

The JavaScript object, `document.cookie`, contains a csv string with all cookies and values. So, we need a utility function, `getCookie`, to extract a specific cookie from the string and return its value.

```
<!DOCTYPE html>
<head>
  <script>
    function getCookie(name)
    {
      var nameEQ = name + "=";
      var ca = document.cookie.split(';');
      for(var i=0;i < ca.length;i++)
      {
        var c = ca[i];
        while (c.charAt(0)==' ') c = c.substring(1, c.length);
        if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length,c.length);
      }
      return null;
    } //getCookie
  </script>
</head>
```

```

<html>
<body>
  <h3>Invoice</h3>
  <label id="lbl_amount"></label>

  <script>
    document.onload = GetAmount();

    function GetAmount()
    {
      amount = getCookie("amount");
      document.getElementById('lbl_amount').innerHTML = amount;
    }

  </script>
</body>
</html>

```

Further reading:

- <https://javascript.info/cookie>
- <https://bordermedia.org/blog/javascript-cookie-not-working-other-page>

4.2.3 Session variables

To enable session variables in Django, we have to add three things (maybe they are there already):

1. In settings.py:

```

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

```

2. In settings.py:

```

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

```

3. Django session variables are saved in a database table, `django_session`. So, we have to migrate the application to create the table in the database if it does not exist already:

```

..\env_clothing\pr_clothing>python manage.py migrate

```

Now, we can create a session variable in a view and access it in a template like this.

View

```

request.session['payment_method'] = "EFT"

```

Template

```

<body>
  Please pay with {{ request.session.payment_method }}.
</body>

```

You can also look at the example in the previous chapter again for the usage of session variables.

Further reading:

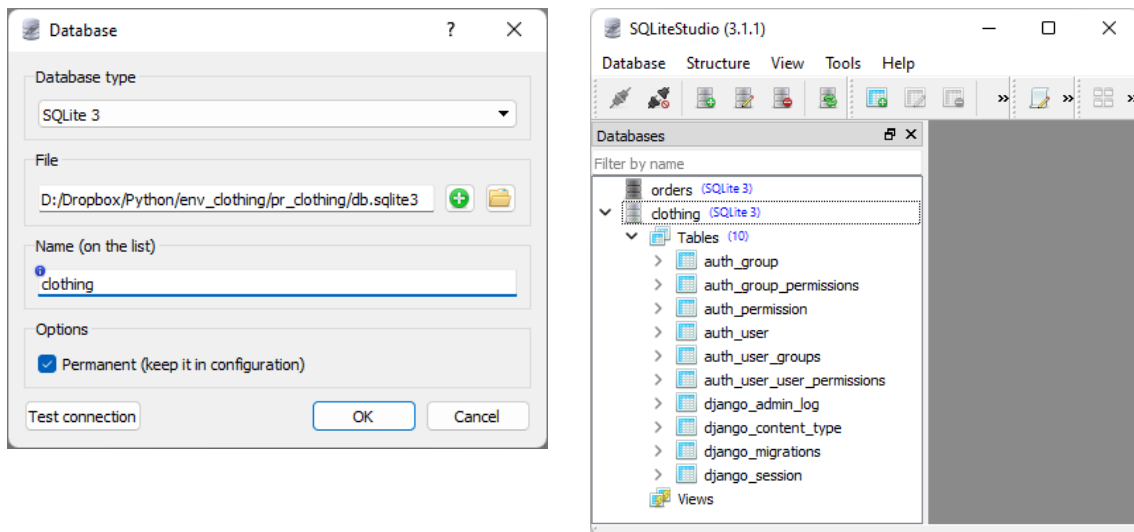
- <https://docs.djangoproject.com/en/5.0/topics/http/sessions/>
- <https://data-flair.training/blogs/django-sessions/>

4.3 Example scenario

The examples in the rest of this tutorial will be based on the following scenario.

A clothing shop wants to keep record of its products. Every product is supplied by a supplier. Customers buy products from the shop. All sales must be recorded.

Follow the procedure explained in Chapter 1 to create a new web app: `env_clothing`, `pr_clothing`, `app_clothing`. Add the database to the list of databases in SQLite Studio and check that the admin tables were created.



4.4 Database options

The bulk of the data is stored in a database on the same or another server as the one on which the web app is hosted. For this course we assume that you have prior database training. If you are not familiar with the basic theory of databases, you should revise it. There is some basic information in Appendix C.

SQLite will suffice for this course. For larger projects, you might need an industrial database environment such as MySQL, PostgreSQL, or MariaDB.

4.4.1 SQLite

The default database when you create a new web app with Django is SQLite. This is a lightweight, file-based, database that resides in the outer project app.

You can use SQLite Studio (<https://sqlitestudio.pl/>) or DBeaver (<https://dbeaver.io/>) to open and inspect this database. Both are free. DBeaver has a nice diagramming tool, but I find the procedure to update table definitions a bit tricky. So, my advice is to use SQLite Studio to create the database and use DBeaver only to create an ERD.

When publishing, you only need to copy the file to the server.

4.4.2 MySQL server

(Optional)

If you want to work with MySQL, you will have to install a local database server. The structure and content of this database can be transferred to an online server when you publish.

You can download the installer from here: <https://dev.mysql.com/downloads/installer/>. There is some guidance in Appendix C.

Use MySQL Workbench as DBMS.

4.4.3 Others

PostgreSQL or MariaDB or SQL Server.

4.4.4 settings.py

We need to specify the database settings.py. For SQLite, it looks like this:

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

BASE_DIR refers to the outer project folder. So, the database resides as ../env_clothing/pr_clothing/db.sqlite3.

For MySQL, the DATABASE entry looks something like this. For this example, the database resides on a MySQL installation on the same PC, but of course you can change it to point to an installation on a webserver.

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "p13",
        "USER": "root",
        "PASSWORD": "*****",
        "HOST": "localhost",
        "PORT": "3306",
        "OPTIONS": {
            "init_command": "SET sql_mode='STRICT_TRANS_TABLES'",
        }
    }
}
```

4.5 Keys and Referential integrity

A **primary key** is a field that will identify a specific record. Primary keys may not be duplicated. It is possible to use the combination of more than one field as primary key. For example, in the Sale table, there may be more than one record with the same customer number and more than one record with the same product code, but the combination of customer number and product code will be unique.

(Emmm ... what happens if a customer comes back the next day and buys another product of the same type? This design does not allow for that possibility.)

Referential integrity is a term used to indicate that no values may exist in the secondary table of a relationship if there is not a corresponding value in the primary table. For instance, we may not add a new record to `Sale` if the corresponding product is not yet registered in `Products`.

A **foreign key** is a field that can be used to identify a record in a primary table in a secondary table. For example, it is possible to determine the name of supplier of a specific product since we have the supplier code in the `Products` table. Foreign keys denote (mostly) a one-to-many relationship: In our design, one supplier can supply more than one product, but a specific product comes from one supplier only.

When defining relationships, we need to indicate what should happen with records in a secondary table if the parent record in the primary table is updated or removed.

On Update: Cascade. This means that if we change the primary key in the primary table, the foreign keys in all related records in the secondary table must also be changed. For example, if the code of a supplier in the `Suppliers` table changes, the supplier codes of all related products should be updated - else, we might end up with products for which the supplier is unknown.

On Delete: Restrict. This will mean that we cannot delete a record in the primary table if there are still related records in a secondary table. For example, we should not be able to delete a supplier if there are still products for that supplier in the database. If we mark **Cascade**, all products will be deleted if a supplier is removed which might mean disaster for a business with old stock on hand if a supplier goes out of business.

4.6 The ORM

The Django web framework includes a default API that allows us to add, delete, modify, and query objects in the underlying database. This API, the so-called **object-relational mapping layer** (ORM), can be used to interact with data from various relational databases such as SQLite, PostgreSQL, and MySQL. The ORM provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries.

Reference: <https://www.scaler.com/topics/django/django-orm-queries/>

4.6.1 The role of models in Django

The models in Django provide a single access point to all data. We first create the models and then migrate the database to the underlying database. See Appendix F for full details of the models in the example application.

A model is a class, defined in Python, that provides an interface between some views and the database. Django takes care of the background stuff so that the programmer does not have to worry about database connections or issue SQL commands to query or update the database. The file `models.py` contain several such classes.

The models in Django provides a single access point to all data. The design of the database, and all CRUD operations can be made through the models.

“A model is the single, definitive source of information about your data. It contains the essential fields and behaviours of the data you’re storing. Django follows the DRY principle. The goal is to define your data model in one place and automatically derive things from it.”

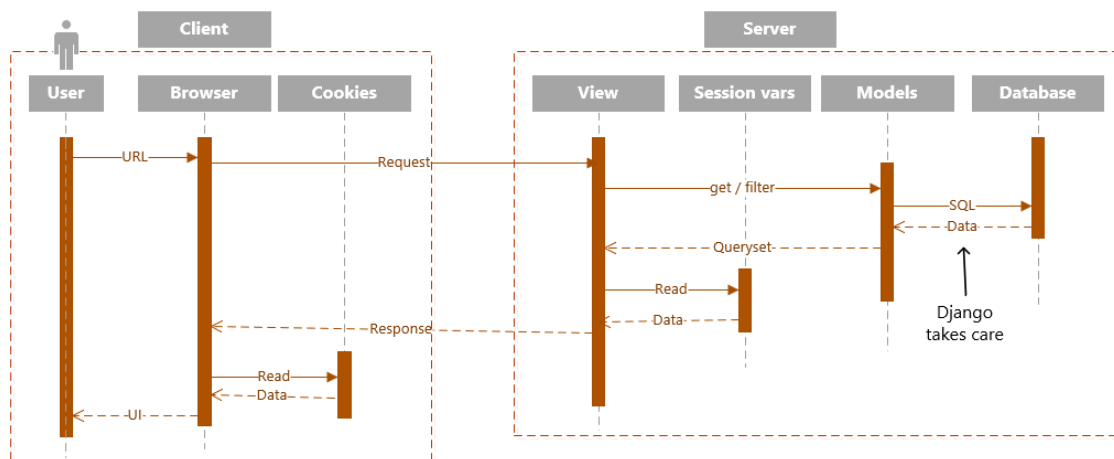
(<https://docs.djangoproject.com/en/4.1/intro/tutorial02/>)

Further reading

- <https://django.readthedocs.io/en/stable/topics/db/models.html>
- <https://devdocs.io/django~4.1/topics/db/models>
- <https://docs.djangoproject.com/en/4.1/#the-model-layer>
- <https://docs.djangoproject.com/en/4.1/ref/models/>
- <https://docs.djangoproject.com/en/4.1/ref/models/fields/>
- <https://devdocs.io/django~4.1/topics/db/queries>

4.6.2 Sequence diagram

The sequence diagram below includes the models as an interface between the views and the database.



Interactions between client and server storage through the models

4.6.3 Coding style / Conventions

This is the official Django style document:

<https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/>

This document lists some best practice guidelines for Django:

<https://learndjango.com/tutorials/django-best-practices-models>

We should adhere to these documents as far as possible. Specifically, remember that Python is case sensitive and therefore by implication Django as well. So, in models and views, table and field names are case sensitive even if the corresponding field name in the database is not.

From learndjango: *"Models should always be capitalized (eg. University, User, Article) and singular (eg. University not Universities) since they represent a single object, not multiple objects."*

From the style document: *"Field names should be all lowercase, using underscores instead of camelCase."* So, `FirstName`, `First_Name`, etc. is not good. Better is `first_name`.

Models are classes. In Python, use CamelCase for class names.

“Start each word with a capital letter. Do not separate words with underscores.”

(<https://realpython.com/python-pep8/#naming-conventions>).

4.6.4 Synchronisation between models and the database

If you open the `models.py` file now, you will notice that it is probably empty.

4.6.4.1 Migration

Any addition, removal or edit to `models.py` should be migrated to the underlying database:

```
..\env_orders\pr_orders>python manage.py makemigrations
..\env_orders\pr_orders>python manage.py migrate
```

4.6.4.2 Modify an existing table

If you want to change a model and update the database:

- Check in the folder `../pr_clothing/app_clothing/migrations`. If there is no `initial.py` file:
 - Ensure manually that the database and models are identical.
 - Create an initial migration for the models as they are at the moment.
`..\env_clothing\pr_clothing>python manage.py makemigrations`
 - Now you have a starting point on which changes will be based.
- Change the model in `models.py`.
- Make sure that `managed=True`.
- Save the file `models.py`.
- Create migrations for the changes from the initial migration:
`..\env_clothing\pr_clothing>python manage.py makemigrations`
- Migrate to the database:
`..\env_clothing\pr_clothing>python manage.py migrate`
- Use SQLite Studio and check that the changes were applied to the database.

4.6.4.3 Out of sync

If your database and the models become out of sync, you might struggle to run the migrations.

If we assume that `models.py` is up to date and correct, you can delete all tables from the database and migrate everything from the beginning to the database. **You will, of course, lose all data.**

- In `models.py`:
 - Ensure that all is fine as you want the database to look like.
 - Delete all admin tables from `models.py`.
 - Make sure that `managed = True` for all tables
 - In your database environment, e.g. SQLite Studio, delete all tables, including the admin tables. You will have to start with all secondary tables or else foreign key restrictions might prevent you from deleting a table.

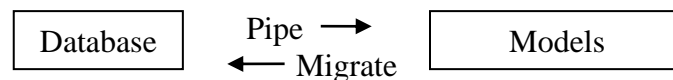
- Delete all migrations under `../pr_clothing/app_clothing/migrations`. Leave `__init__.py`.
- In the terminal window:
`..\env_clothing\pr_clothing>python manage.py makemigrations`
`..\env_clothing\pr_clothing>python manage.py migrate`

Further reading:

- <https://stackoverflow.com/questions/43880426/how-to-force-migrations-to-a-db-if-some-tables-already-exist-in-django>
- <https://www.linkedin.com/pulse/how-do-i-reset-django-migration-nitin-raturi>

4.6.4.4 From DB to models

It is possible, although not recommended, to design the database first and then use that as basis for the models. This process is referred to as piping.



- To see how the models will look like when created:
`(..) ~\pr_clothing $ python manage.py inspectdb [table name]`
- To create the models in `models.py`:
`(..) ~\pr_clothing $ python manage.py inspectdb [table name]`
`> app_clothing/models.py`
- **It is recommended to use inspectdb without piping. Copy the model from the terminal and paste into models.py.**
- Inspect the content of `models.py` and make sure that all is fine.

4.6.5 Register models in `admin.py`

This is only needed for the admin module of Django. We will provide details when we discuss user administration in a later chapter.

4.7 Model syntax

4.7.1 Field types

Below is an example of a simple model.

```

class Person(models.Model):
    person_id = models.AutoField(primary_key=True)
    id_number = models.TextField(blank=True, null=True)
    name_surname = models.TextField()
    address = models.TextField(blank=True, null=True)
    email = models.CharField(max_length=100, blank=True, null=True)

    dob = models.DateField(blank=True, null=True)
    is_adult = models.BooleanField(blank=True, null=True)
    gender = models.TextField(blank=True, null=True)

    cv_file = models.FileField(db_column='cv_file_name', upload_to='files/', blank=True, null=True)
    photo_file = models.ImageField(db_column='photo_file_name', upload_to='images/')

    class Meta:
        managed = True
  
```

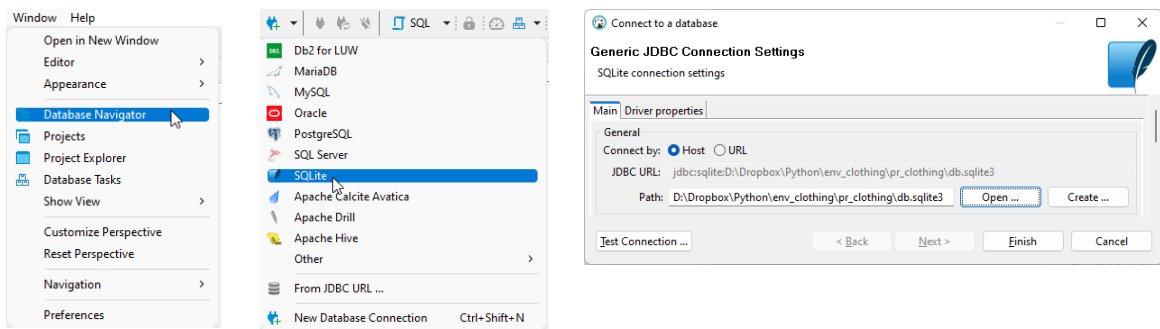
```
db_table = 'Person'
```

- AutoField: Auto-number field used for primary keys. Note that if you do not explicitly create a primary key, Django will create a primary key field, `id`, in the background. Database type integer.
- TextField: Used for text entries. Database datatype
- CharField: Use for text entries with specified length. Database type varchar
- Other possible field types:
 - DateField, DateTimeField, IntegerField, DecimalField, BooleanField, FileField, ImageField
- The class `Meta` is a subclass of the other classes. Note the indenting.
 - `managed = False` means that subsequent changes to the model will not be applied to the underlying database during migrations.
 - `managed = True` means that subsequent changes to the model will be applied to the underlying database during migrations.

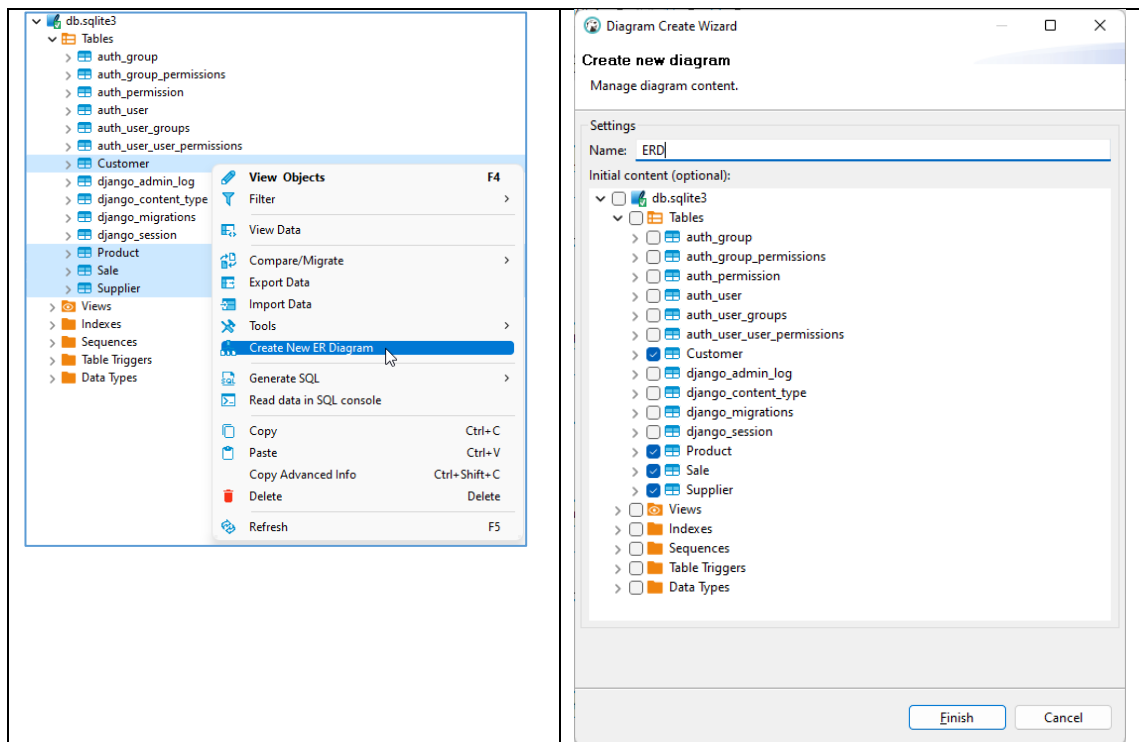
4.7.2 Visualising the design

Once you migrated the models to the underlying database, you can draw an ERD of the design. Unfortunately, SQLite Studio does not have a diagramming tool, but DBeaver has.

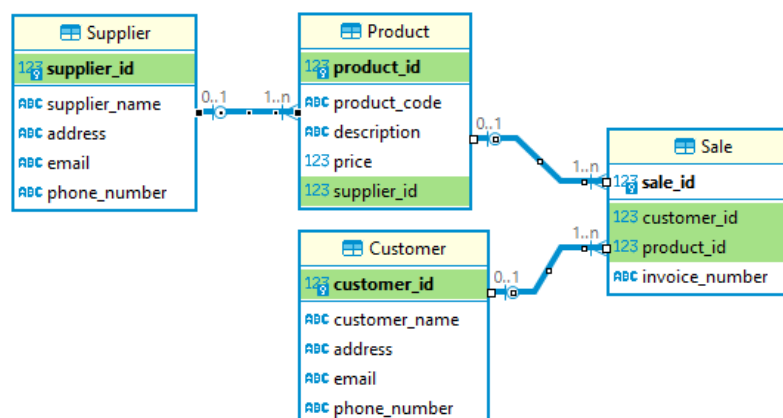
- Open DBeaver and make sure that the Database Navigator window is visible.
- Add the clothing database.



- Right-click on the database name and select `View diagram`. A tab, `ER Diagram`, will open and the existing tables will be added automatically. This is somewhat cluttered with all the admin tables.
- Instead, expand the `Tables` group and select only the tables that we added above. Then right-click and select `Create New ER Diagram`. Give the new diagram a name and click `Finish`.



- If the foreign keys were added correctly, the links should be drawn automatically. Save the diagram (File/Save).
- Hints:
 - It is good practice to set up an ERD such that the one-to-many relationships are read from left to right. Also, relationships should not cross over. This allows for easier readability and maintenance.
 - Right-click on the diagram and change the notation to Crows Foot.
 - The data types are indicated through the icons to the left of the column name. ABC is TEXT and 123 is numeric.
 - Key fields are indicated with small key symbol to the left of the column name.
 - If you select a relationship, the columns that are involved are highlighted.



4.7.3 Relationships between models

When thinking about relationships in Django, we should not be biased by our experience with relationships in databases. Relationships in Django models are handled somewhat differently.

Although the relationship is established through the foreign keys behind the scenes, in a Django model we refer to the full parent object in a child model. See the repex example.

4.7.4 Foreign keys

4.7.4.1 One-to-Many relationships

Consider the following two models:

```
class Supplier(models.Model):
    supplier_id = models.AutoField(primary_key=True)
    user = models.OneToOneField('AuthUser', models.DO_NOTHING) #, null=True, blank=True)
    supplier_name = models.TextField(blank=True, null=True)
    address = models.TextField(blank=True, null=True)
    phone_number = models.TextField(blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'Supplier'

class Product(models.Model):
    product_id = models.AutoField(primary_key=True)
    product_code = models.TextField(blank=True, null=True)
    description = models.TextField(blank=True, null=True)
    description_file = models.FileField(db_column="description_file_name", upload_to='files/')
    image_file = models.ImageField(db_column="image_file_name", upload_to='images/')
    price = models.DecimalField(max_digits=10, decimal_places=5, blank=True, null=True)
    supplier = models.ForeignKey('Supplier', models.DO_NOTHING, blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'Product'
```

- There is a one:many relationship between Supplier and Product. That means that a supplier may supply many products, but any specific product is supplied by exactly one supplier.
- The Product model contains a field supplier. In the database, this is represented with the foreign key, supplier_id in the Product table. An entire record is referenced in the model – not only the key field of the primary table.
- In a view, reference to a column in a foreign table is done through '___' (double underscore), for example product__supplier__supplier_name.
- In a template tag, reference to a column in a foreign table is done though '.' (period), for example product.supplier.supplier_name.

References

- https://docs.djangoproject.com/en/5.0/topics/db/examples/one_to_one/
- <https://vegibit.com/how-to-filter-foreign-key-in-django/>
- <https://docs.djangoproject.com/en/5.0/ref/models/queriesets/#prefetch-related>
- <https://stackoverflow.com/questions/9176430/django-does-prefetch-related-follow-reverse-relationship-lookup>

4.7.4.2 Many-to-Many relationships

In our example, a type of product can be bought by more than one customer and a customer can buy more than one product. So, there is a many_many relationship between Customer and Product. We use an intermediate table, Sale, with two 1:M relationships, namely Customer:Sale and Product:Sale.

```

class Sale(models.Model):
    sale_id = models.AutoField(primary_key=True)
    customer = models.ForeignKey(Customer, models.DO_NOTHING, blank=True, null=True)
    product = models.ForeignKey(Product, models.DO_NOTHING, blank=True, null=True)
    invoice_number = models.TextField(blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'Sale'
        unique_together = (('customer', 'product'),)

```

- It is important to note that we cannot have combined primary keys in a Django model. We use an AutoField for the key and specify that the combination of customer and product should be unique through the unique_together attribute in the Meta class.

4.7.5 Methods

In Python, functions that belong to a class are referred to as methods.

You can add methods to the model classes to return a data field in a specific format or return manipulated versions of a data field.

4.7.5.1 vat()

In the class Product, you can add a method for VAT.

```

from decimal import Decimal

class Product(models.Model):
    ...

    def vat(self):
        return self.price * Decimal('0.15')

    def __str__(self):
        return self.product_code + ' ' + self.description

```

Notes:

- Before we can multiply a float with a decimal, we have to convert the float to decimal using decimal.Decimal.
- Note that method names must be written in lower case. Separate words with underscores.
- Remember to call functions and methods with ().

4.7.5.2 Constructor

Don't use `__init__()` in Django. See this for an explanation and alternative:
<https://docs.djangoproject.com/en/5.0/ref/models/instances/#creating-objects>.

Rather, add a class method like this:

```

class Customer(models.Model):
    customer_id = models.AutoField(primary_key=True)
    customer_name = models.TextField()
    address = models.TextField(blank=True, null=True)
    email = models.TextField(blank=True, null=True)
    phone_number = models.TextField(blank=True, null=True)

```

```

class Meta:
    managed = True
    db_table = 'Customer'

@classmethod
def create(cls, name):
    customer = cls(customer_name=name)
    return customer

def __str__(self):
    return str(self.customer_id) + ' ' + self.customer_name

```

Usage:

```

customer = Customer.create("Mike Matthews")
customer.save() #Or else the auto-numbered id will not be created.

```

4.7.5.3 __str().__

The `__str().__` method defines a human-readable representation of the model. It is called by the built-in `print().__`, `str().__`, and `format().__` functions. This is the equivalent of the C# `ToString().__`.

In this case, `str(customer)` will return the customer id concatenated with the customer name.

Further reading:

- <https://docs.python.org/3/reference/datamodel.html#special-method-names>

4.7.6 `select_related` and `prefetch_related` (Enrichment)

4.7.6.1 Why `select_related` and `prefetch_related`?

Question to ChatGPT:

Why do we need `select_related` if the parent record is implicitly included in a child model?

Answer:

In Django, when dealing with related models, there are two main ways to retrieve related objects: `select_related` and `prefetch_related`. Both serve optimization purposes, but they work differently.

When you access a related object in Django, let's say a `ForeignKey` or a `OneToOneField`, Django follows a lazy-loading approach by default. This means that when you access the related object, Django makes a separate database query to fetch that related object. For example, if we assume that `child` is an instance of a model with a `ForeignKey` to `parent`:

```
related_parent = child.parent
```

Here, `related_parent` is fetched with a separate database query, resulting in potential performance issues, especially if you're accessing related objects in a loop or in a large dataset.

However, if you know in advance that you will need the related object(s) and want to minimize database queries, you can use `select_related`. `select_related` works by performing a single SQL query to retrieve the primary object and its related objects in one go. This is particularly useful when you're fetching a single related object or a small set of related objects.

For example, this retrieves all `Child` objects and their related `Parent` objects in a single query:

```
children = Child.objects.select_related('parent').all()
```

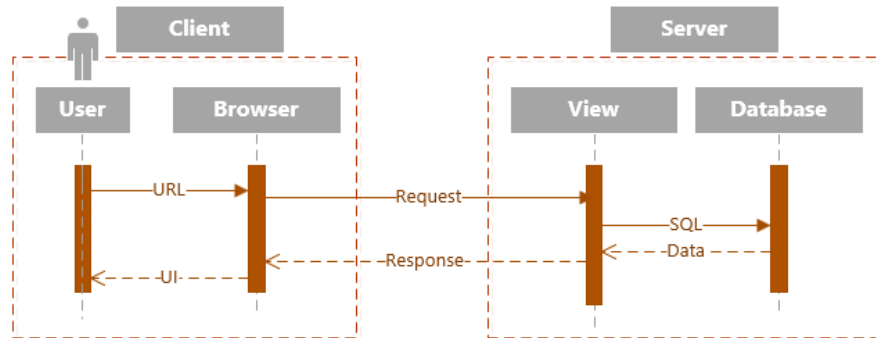
This way, when you access the related object later, there's no need for an additional database query because the related objects are already fetched.

So, to answer your question, while accessing the parent record is implicitly included in the child model, using `select_related` ensures that this is done efficiently in a single database query rather than triggering separate queries for each related object, thus improving performance in scenarios where related objects need to be accessed frequently.

4.8 Direct access (i.e. not using models)

Although it is possible for a view to access a database directly from a view through an SQL query, it is not advised. It is preferable to access the database through the models.

The sequence diagram below shows the sequence of events in case you access the database directly.



The example below shows a form that can be used to execute SQL queries.

4.8.1 Paths

```
from django.urls.conf import path, re_path
from app_clothing.views.vw_home import index
from app_clothing.views.vw_sql import sql, sql_post

urlpatterns = [
    path('', index, name='index'),
    re_path(r'(?i)index/', index),

    re_path(r'(?i)sql/', sql),
    re_path(r'(?i)sql_post/', sql_post),
]
```

4.8.2 Templates

4.8.2.1 index.html

```
{% extends "base.html" %}
{% block content %}
    <h2>Clothing shop</h2>
    <ul>
        <li><a href="/SQL/">SQL</a></li>
    </ul>
{% endblock %}
```

4.8.2.2 sql_form.html

```
{% extends "base.html" %}
{% block content %}
<form action="/sql_post/" method="post">
    {% csrf_token %}
    <table>
        <tr>
            <td><b>SQL</b></td>
            <td><input name="sql" size="50" value="{{sql}}"></td>
        </tr>
        <tr>
            <td><br></td>
        </tr>
    </table>
</form>
```

```

        </tr>
        <tr>
            <td colspan="2" align="center">
                <input type="submit" value="Submit">
                <input type="button" value="Cancel" onclick="window.location='/';">
            </td>
        </tr>
    </table>

    {% if "SELECT" in sql %}
        <table>
            <!-- Headers -->
            <tr>
                {% for col in cols %}
                    <td align="left"><b>{{ col }}</b></td>
                {% endfor %}
            </tr>

            <!-- Data rows -->
            {% for row in rows %}
                <tr>
                    {% for col in row %}
                        <td>{{ col }}</td>
                    {% endfor %}
                </tr>
            {% endfor %}
        </table>
    {% endif %}

    {% if error %}
        {{ error | linebreaksbr }}
    {% endif %}

</form>

{% endblock %}

```

Notes

- The first table is used for layout of the main part.
- The second table is used for the results

SQL <input type="text"/>
<input type="button" value="Submit"/> <input type="button" value="Cancel"/>

SQL	SELECT * FROM Supplier			
<input type="button" value="Submit"/> <input type="button" value="Cancel"/>				
supplier_id	supplier_name	address	email	phone_number
1	Supplier 1	None	None	None
2	Supplier 2	None	None	None
3	Supplier 3	None	None	None
4	Supplier4	None	None	None
5	Supplier 5	None	None	None
8	Supplier 5	None	None	None

- The last section is used to display an error message – if any.

4.8.3 Views

4.8.3.1 vw_home.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

def index(request):
    return render(request, 'index.html')

def show_message(request, message):
    return render (request, "messages/message.html", { "message": message})
```

4.8.3.2 vw_sql.py

```
from django.shortcuts import render
from . import vw_home
from django.db import connection

def sql(request):
    return render(request, 'sql_form.html')

def sql_post(request):
    if request.method == "POST":
        sql = request.POST['sql']
        cursor = connection.cursor()
        try:
            cursor.execute(sql)
            if "SELECT" in sql:
                columns = [col[0] for col in cursor.description]
                rows = cursor.fetchall()
                return render(request, 'sql_form.html', \
                    {'sql': sql, 'cols': columns, 'rows': rows})
            else:
                return render(request, 'sql_form.html', {'sql': sql})
        except Exception as e:
            return render(request, 'sql_form.html', {'sql': sql, 'error': e})
    return vw_home.index(request)
```

Notes

- The import of connection.
- If the sql is a SELECT statement, the page is rendered with the results with column headers and rows.
- If the sql is an UPDATE, INSERT or DELETE, the page is rendered without results.
- If the SQL statement is in error, the page is rendered with the error message.
- The last return statement is a fall-through in case the user succeeded to call this function without issuing a submit.

4.9 API shell

We can access data in the API shell through the models. Note that this is Python, and all text is strictly case sensitive.

<https://docs.djangoproject.com/en/4.1/intro/tutorial02/>

4.9.1 Start the shell

Start the API shell

```
$ python manage.py shell
```

Optional: To get an interactive interpreter, you can do this once before starting the shell:

```
$ pip install ipython
```

To clear the shell:

```
>>> clear
```

To exit the shell: Ctrl-Z or exit()

4.9.2 Import the relevant model classes

```
>>> from app_clothing.models import Customer, Supplier, Product, Sale
```

4.9.3 Some specific API examples

List all objects in a table:

```
>>> Customer.objects.all()
```

Add an object

```
>>> c = Customer(customer_id=4, customer_name="Customer 4")
>>> c.save()
```

Access a specific object

```
>>> c = Customer.objects.all()[2]
>>> c.customer_name
'Customer 3'
```

If a `__str__()` method is available:

```
>>> c
<Customer: Customer>
```

Find a specific object:

```
>>> q = Customer.objects.filter(telephone="123")
>>> c = q[0]
>>> c.customer_name
'Customer 1'
```

Find by primary key:

```
>>> q = Customer.objects.get(pk="C3")
```

4.9.4 Debugging a module

Suppose you call the function `vat()` in the `Product` class to see if it works correctly. If you change the function and then call the function again from the shell, you will not see the effect of the changes. You must unload and then reload models.

4.9.4.1 Using sys.modules

Run first time:

```
>>> import sys
>>> from app_clothing.models import Product
>>> Product.objects.all()[0].vat()
```

Inspect output

repeat:

Make changes to the vat method and save.

Run again:

```
>>> del sys.modules['app_clothing.models']
>>> from app_clothing.models import Product
>>> Product.objects.all()[0].vat()
```

Inspect output

until satisfied

Reference: <https://stackoverflow.com/questions/6946376/how-to-reload-a-class-in-python-shell>

4.9.4.2 Using reload

Run first time:

```
>>> from importlib import reload
>>> import app_clothing.models
>>> from app_clothing.models import Product
>>> Product.objects.all()[0].vat()
```

Inspect output

repeat:

Make changes to the vat method and save.

Run again:

```
>>> reload(app_clothing.models)
>>> from app_clothing.models import Product
>>> Product.objects.all()[0].vat()
```

until satisfied

Reference: <https://stackoverflow.com/questions/6946376/how-to-reload-a-class-in-python-shell>

4.10 References

<https://docs.djangoproject.com/en/4.1/intro/tutorial02/>
<https://help.pythonanywhere.com/pages/FollowingTheDjangoTutorial/>
https://www.w3schools.com/django/django_models.php
<https://help.pythonanywhere.com/pages/UsingMySQL/>
<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Models>

4.11 Summary

In this chapter, we introduced the various data storage facilities in Django. Data can be saved temporarily for a browser session in the session variables or permanently in a database. We indicated that the primary method for interacting with data in Django is through the ORM. We presented an example of accessing data in web app through direct SQL queries and showed how data can be manipulated in the Python shell.

In the next chapter, we will show how to provide front-end access to data through the models.

Chapter 5: Client access to data

<https://docs.djangoproject.com/en/4.1/intro/tutorial04/>
<https://django.readthedocs.io/en/stable/intro/tutorial04.html>
<https://django.readthedocs.io/en/stable/faq/index.html>
<https://docs.djangoproject.com/en/4.1/#forms>
<https://docs.djangoproject.com/en/4.1/ref/forms/>
<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Forms>
<https://devdocs.io/django~4.1/topics/forms/modelforms>

5.1 Introduction

In this chapter, we will present various examples of how to retrieve and update data in a Django web app.

5.2 Python collections

- A **list** is an ordered collection of items surrounded by []. It is similar to an ArrayList in C#. Every item can be an integer, string, list, or dictionary. Items in a list can be accessed by index between square brackets.
Example: `list1 = [1, 4, "John", product]`
`list[1] = 4`
Methods: `append()`, `pop()`
- A **set** is an unordered collection of items written between { }. Indexing is not possible. Duplicates are not allowed.
Example: `set1 = { 1, 2, 3, "John", product, supplier }`
Methods: `add()`, `pop()`
- **Dictionaries** are key/value pairs surrounded by { }. The context parameter in the render method is a dictionary. Every value element can, in turn, be a string, integer, dictionary, list or set.
Example `{ "suppliers": suppliers, "products": products }`
Methods: `update()`, `pop()`
- Further reading
 - <https://testbook.com/key-differences/difference-between-list-tuple-set-and-dictionary-in-python>

5.3 QuerySet

5.3.1 What is a QuerySet?

The Django class, `QuerySet`, forms the basis of Django ORM model and it is imperative that we understand how to manipulate QuerySets.

A `QuerySet` is a set – not a list. It is unordered and cannot be indexed. It is a collection of data from a database and consists of a list of objects. `QuerySet` is a Django-specific data type and does not share the same methods as Python sets. QuerySets allow filtering and ordering of data at an early stage.

Further reading:

- https://www.w3schools.com/django/django_queryset.php
- <https://docs.djangoproject.com/en/4.1/ref/models/querysets/>
- <https://ctrlzblog.com/django-queryset-filter-15-examples/>
- <https://www.programink.com/django-tutorial/django-queryset.html>
- <https://stackoverflow.com/questions/58638352/queryset-v-s-list-django>

5.3.2 QuerySet methods

If Customer, Supplier and Product are models, then

- `Customer.objects.all()`: Returns all Customer objects as a QuerySet.
- `Customer.objects.get(customer_id=1)`: Returns a single Customer object.
- `Customer.objects.order_by('customer_name')`: Returns objects in the order specified by the parameter.
- `Product.objects.filter(supplier__supplier_id = 3)`: Returns a subset of Product as specified by the condition – in this case all products of supplier with id = 2.
 - Note the double underscore to refer to the foreign key object followed by the foreign key object's key value.
- `Supplier.objects.values()`: Returns a QuerySet of dictionaries.
 - Each object is returned as a dictionary with the field names and values as key/value pairs.
- `Supplier.objects.values_list('supplier_name', 'email')`: Returns a QuerySet of tuples.
 - Returns only columns that are specified.
 - Hint: Use the `flat=True` parameter when only one field name is specified.
`Supplier.objects.values_list('supplier_name', flat=True)`

References:

- <https://docs.djangoproject.com/en/4.1/ref/models/querysets/#get>
- https://www.w3schools.com/django/django_queryset_filter.php
- <https://docs.djangoproject.com/en/4.1/ref/models/querysets/#filter>
- https://www.w3schools.com/django/django_queryset_get.php
- <https://ctrlzblog.com/django-queryset-filter-15-examples/>

5.3.3 Field lookup (`__gt`, `__lt`, `__gte`, `__lte`, `__in`, `__contains`, etc.)

- <https://dev.to/vincod/django-queryset-filters-gt-lt-gte-lte-13d9>
- https://www.w3schools.com/django/django_ref_field_lookups.php
- https://www.w3schools.com/django/ref_lookups_gte.php
- https://www.w3schools.com/django/ref_lookups_contains.php
- <https://ctrlzblog.com/django-queryset-filter-15-examples/>

5.3.4 Combining QuerySets (AND, OR, etc)

- <https://docs.djangoproject.com/en/4.1/ref/models/querysets/#operators-that-return-new-querysets>

Instead of `'|'`, rather consider `__in` as in

https://www.w3schools.com/django/ref_lookups_in.php.

5.3.5 Filter foreign keys

- <https://vegibit.com/how-to-filter-foreign-key-in-django/>
- <https://docs.djangoproject.com/en/5.0/ref/models/queries/#prefetch-related>

5.4 Data retrieval: Examples

5.4.1 Example 1: Render content without a template

In this example, we will display a comma-separated list of customer names in the database.

5.4.1.1 Create the view

Edit the `views.py` (or `vw_customers.py`) file to render responses directly (not via a template file). The formatting is done in the view and the final result is sent to the browser directly.

```
from django.http import HttpResponse
from .models import Customer

def Customers(request):
    customers = Customer.objects.all().order_by('customer_name')
    output = ', '.join([c.customer_name for c in customers])
    #output = ', '.join([str(c) for c in customers])
    return HttpResponse(output)
```

Notes

- The first line in the view returns a queryset and assigns it to a variable.
- The second line steps through all customer names in the query set and join them in a comma-separated string.
- The third line is commented out. It provides an alternative to list the customers by using the `__str__()` function as defined in the model.

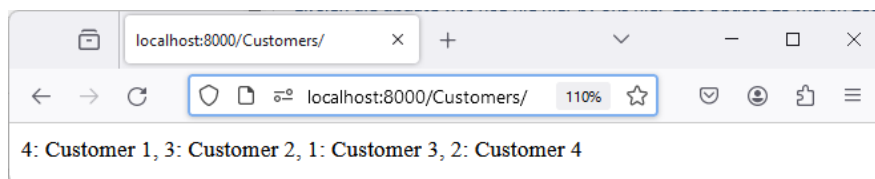
5.4.1.2 Path

Edit the file `../pr_clothing/urls.py` to include a path to `vw_customers.customers`:

```
from app_clothing.views.vw_customers import customers
urlpatterns = [
    ...,
    path('customers/', customers),
]
```

- Note the trailing forward slash after the route. There are no leading slashes in the `urlpatterns` list.
- I created a separate file for all customer-related views. So, we need to import the views from there.

5.4.1.3 Test



- You should see a comma-separated string of customer names. If you do not see anything, check that there is test data in the database.

5.4.2 Example 2: Render content through a template

In this example, we will return a bulleted list of all products in the database sorted by description.

5.4.2.1 Template

In `pr_clothing/app_clothing/templates`, create a folder, `products`. Add a file, `product_list.html`, to this folder. Remember to add the folder under `TEMPLATES` in `settings.py`.

```
{% if product_list %}
<ul>
  {% for product in product_list %}
    <li><a href="/product_details/{{ product.product_code }}">{{ product.description }}</a></li>
  {% endfor %}
</ul>
{% else %}
  <p>No products are available.</p>
{% endif %}
```

Notes

- `product_list` is a context as provided by the view
- Text between `{{ }}` is treated as a variable name and the value of that variable is printed.
- The linked path is given as a relative path. A relative path is everything after the domain, including the forward slash and directory.
- The linked path refers to a page with all details of the selected product. We will do this in Example 4 below.
- If the `urlpatterns` in `app.urls` include a trailing slash, the linked path should also include the trailing slash.

5.4.2.2 View

Under `views`, create a file `vw_products.py`. Add a view as below. The view obtains the list of objects and sends it through to the template. Formatting is done in the template which will render the content in the browser.

```
from django.shortcuts import render
from app_clothing.models import Product

def product_list(request):
    product_list = Product.objects.order_by('description')
    context = {'product_list': product_list, }
    return render(request, 'product_list.html', context)
```

- `product_list` is a context that is passed to the template.

5.4.2.3 Path

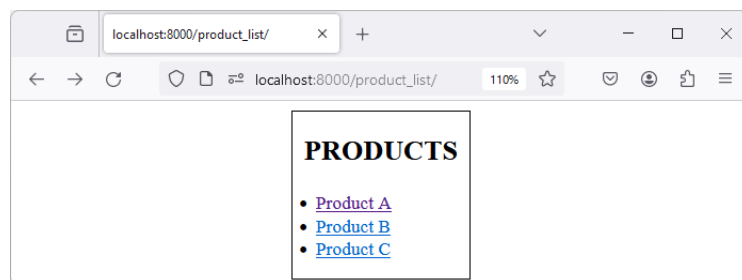
Edit the `../project/app/urls.py` file to include a path to the Products view:

```
from app_clothing.views.vw_products import product_list

urlpatterns = [
    ...,
    path('productlist/', product_list),
]
```

5.4.2.4 Test

Make sure that you have test data in the database.



5.4.3 Example 3: Filter by argument

In this example, we will return a bulleted list of some products in the database.

5.4.3.1 Template

- The template is the same as above

5.4.3.2 View

```
def product_list(request, supplier_id = -1):
    if supplier_id == -1:
        product_list = Product.objects.order_by("description") ()
    else:
        product_list = Product.objects.filter(supplier__supplier_id = supplier_id) \
            .order_by("description")
    context = { 'product_list': product_list, }
    return render(request, 'products.html', context)
```

- Notes
 - The `supplier_id` parameter has a default value which will be assigned if the parameter is not assigned in the URL.
 - The `'\'` character in the code below else indicates that the code line continues on the next line.
 - Note the double underscore to refer to a specific field of the supplier foreign table.
 - The fact that Django uses a foreign table and not a foreign key, allows us to specify any field of the Supplier table.
 - One would expect `==` in the filter for product list which is not the case. This is not general Python, it is Django-specific code.

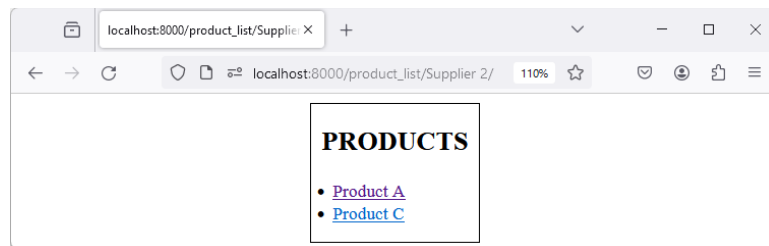
5.4.3.3 Path

```
urlpatterns = [
    ...,
    path('product_list/<int:supplier_id>/', product_list),
]
```

- References

- <https://docs.djangoproject.com/en/4.1/topics/http/urls/#example>
- <https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters>

5.4.3.4 Test



- Provided that you have test data with products with `supplier_name = "Supplier 2"`, you should see a list of all products for this supplier.

5.4.4 Example 4: Find a single object

In this example, we will find a specific product and display all of its attributes. A 404 page will be displayed if the product code is not found.

5.4.4.1 Template

Under `templates\products`, create a new file, `product_details.html`.

```
{% extends "base.html" %}
{% block content %}

<h3>Product details</h3>

<table>
  <tr>
    <td><b>Product code</b></td>
    <td>{{ product.product_code }}</td>
  </tr>
  <tr>
    <td><b>Description</b></td>
    <td>{{ product.description }}</td>
  </tr>
  <tr>
    <td><b>Price</b></td>
    <td>{{ product.price | floatformat:2 }}</td>
  </tr>
  <tr>
    <td><b>Supplier id</b></td>
    <td>{{ product.supplier.supplier_id }}</td>
  </tr>
  <tr>
    <td><b>Supplier name</b></td>
    <td>{{ product.supplier.supplier_name }}</td>
  </tr>
</table>
```



```
<br>
<input type="button" value="Back" onclick="window.location='/product_list/'; ">
<p></p>

{% endblock %}
```

Notes:

- The product variable is sent through as context from the view.
- Make sure that you understand why the Django tags contain the values that they do.
- The Back button will return the user to the list of products.

NB: Read this:

- <https://docs.djangoproject.com/en/4.1/topics/templates/>
- <https://docs.djangoproject.com/en/4.1/ref/templates/language/>

5.4.4.2 View

In `vw_products.py`, add:

```
from django.shortcuts import get_object_or_404

def product_details(request, pcode='P2'):
    product = get_object_or_404(Product, product_code=pcode)
    return render(request, 'product.html', {'product': product})
```

Notes:

- The product variable is sent through as context to the template.

5.4.4.3 Path

Edit the `urls.py` file to include a path to the `product_details` view:

```
from app_clothing.views.vw_products import product_list, product_details

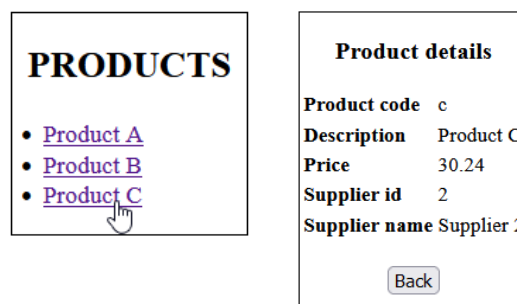
urlpatterns = [
    ...,
    path('product_details/<str:product_code>/', product_details),
]
```

Note:

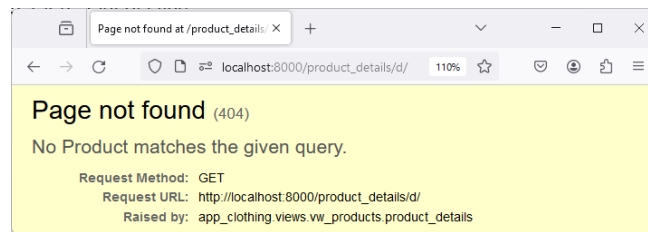
- The trailing slash after the pattern. If there is a trailing slash here, all paths to this view in templates must include the trailing slash.

5.4.4.4 Test

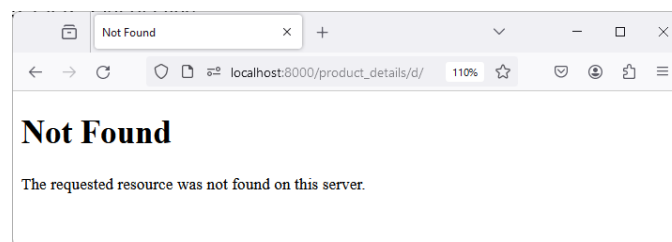
- Click on one of the links in Example 2. You should see something like this:



- If you type the URL directly in the browser with a non-existing product code, you should see this:



- Change `settings.py` so that `DEBUG = False` and test again with a non-existent product code:



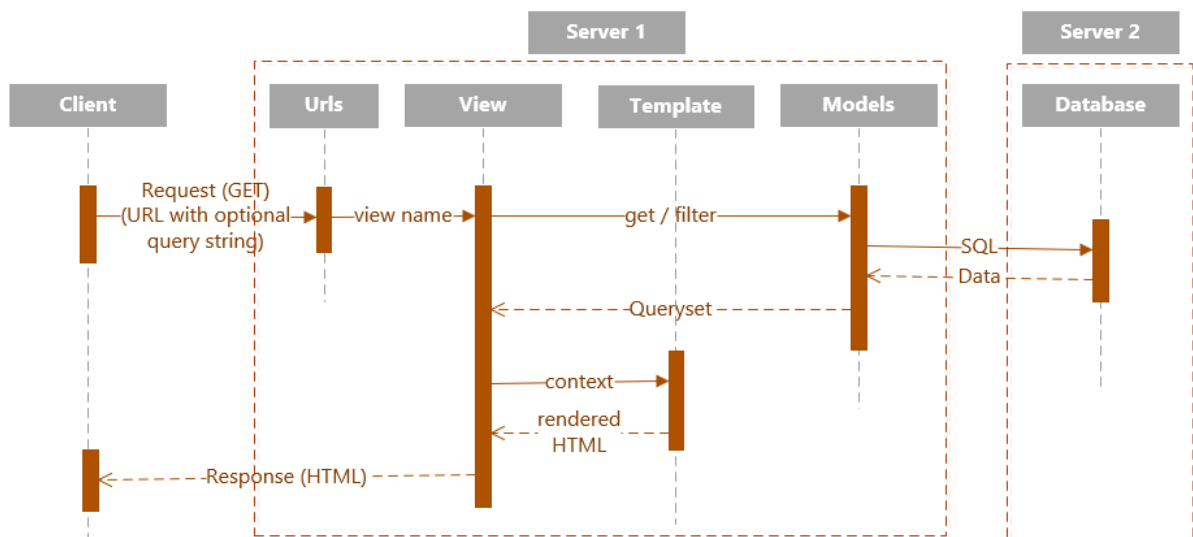
- Change `settings.py` back so that `DEBUG = True` again. You want to see the error messages during debugging. Take note of the security warning in the comment line.

5.4.5 Sequence of events

“Django uses request and response objects to pass state through the system. When a page is requested, Django creates an [HttpRequest](#) object that contains metadata about the request. Then Django loads the appropriate view, passing the [HttpRequest](#) as the first argument to the view function. Each view is responsible for returning an [HttpResponse](#) object.” (<https://docs.djangoproject.com/en/4.2/ref/request-response/>.)

The sequence diagram below shows an example of the process that is followed when a client requests data.

1. The client issues a GET request to the server.
2. The `urls.py` file maps the given URL to a view name with parameters. Remember that views are functions.
3. The view requests the necessary data from the models.
4. The models query the database.
(Note that it is possible but not necessarily so, that the database resides on another server – as in a 3-tier architecture.)
5. The database responds with the requested data to the model.
6. The model sends a `QuerySet` back to the view. This is a set of data that can be provided to the template.
7. The view executes the embedded code in the template and fills the placeholders (`{{ }}` tags) with the requested data (referred to as the context).
8. The view returns an `HttpResponse` object to the client. This object contains the page source that the browser will use to display the page.



Interactions between client and server when the client requests data

5.5 CRUD (Create, Read, Update, Delete)

In this section, we will develop facilities to list all registered suppliers, add new suppliers, edit their details and remove unwanted supplier entries.

5.5.1 Index

Before we can go any further, we need an index from where all navigation will start. Update the index of Chapter 4 to look like this.



5.5.2 List records

In this example, we will list all registered suppliers in a table and add links to add, edit or delete a supplier.

5.5.2.1 Template

Under `../project/app/templates`, create a folder, `suppliers`, and add a new empty file, `supplier_list.html`. Remember to add this folder to `settings.py`.

```

{% extends "base.html" %}
{% block content %}

    <h2>SUPPLIERS</h2>

    <!-- Column headers -->
    <table>
        <tr>
            <th width="70px" align="left">ID</th>
            <th width="120px" align="left">Name</th>
            <th width="200px" align="left">Address</th>
  
```

```

        <th width="120px" align="left">Telephone</th>
        <td><a href="/supplier_add/">Add supplier</a></td>
        <td></td>
    </tr>
</table>

<!-- Data rows -->
<table>
    {% for supplier in supplier_list %}
        <tr>
            <td width="70px">{{ supplier.supplier_id }}</td>
            <td width="120px">{{ supplier.supplier_name }}</td>
            <td width="200px">{{ supplier.address | linebreaksbr }}</td>
            <td width="120px">{{ supplier.phone_number }}</td>
            <td><a href="/supplier_edit/{{ supplier.supplier_id }}">Edit</a></td>
            <td><a href="/supplier_delete/{{ supplier.supplier_id }}">Delete</a></td>
            <td><a href="/product_list/{{ supplier.supplier_id }}">Products</a></td>
        </tr>
    {% endfor %}
</table>
<br>

<input type="button" value="Home" onclick="window.location='/'">

{% endblock %}

```

Notes

- We use two separate tables - one for the headers and one for the data rows. This is a matter of taste - you could also display the data in other ways.
- The model data is obtained from the view as a context object (supplier_list).
- The context data is displayed in Django tags {{ }}.
- The, Add, Edit and Delete links refer to specific paths in the urls.py file. They will be discussed below.
- The Add, Edit and Delete paths are given as relative paths. A relative path is everything after the domain, including the forward slash and directory.
- The Products link returns a list of products for the specific supplier. It follows the procedure of Example 3 in the previous section.
- The <input> field at the end displays a button that will take the user back to the index page.
- The
 tag adds a line space before the button.

5.5.2.2 View

Under views, create a file vw_suppliers.py. Add a view as below.

```

from django.shortcuts import render
from app_clothing.models import Supplier
from . import vw_home

def supplier_list(request):
    supplier_list = Supplier.objects.order_by('supplier_name')
    context = {'supplier_list': supplier_list, }
    return render (request, "supplier_list.html", context)

```

5.5.2.3 Path

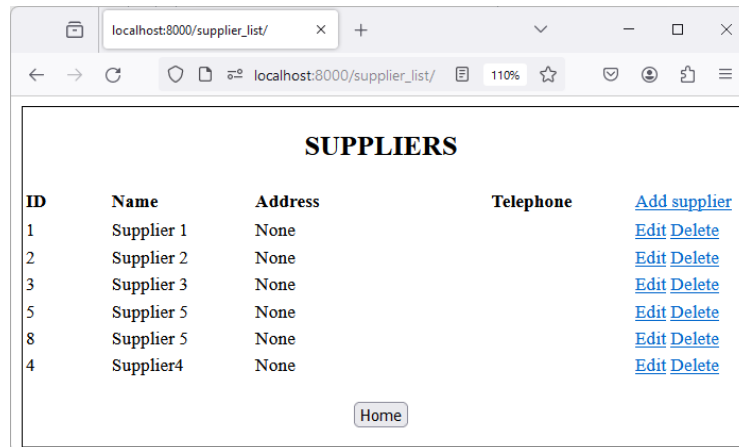
```

from app_clothing.views.vw_suppliers import supplier_list

urlpatterns = [
    ...,
    path('supplier_list/', supplier_list),
]

```

5.5.2.4 Test



ID	Name	Address	Telephone	Add supplier
1	Supplier 1	None		Edit Delete
2	Supplier 2	None		Edit Delete
3	Supplier 3	None		Edit Delete
5	Supplier 5	None		Edit Delete
8	Supplier 5	None		Edit Delete
4	Supplier4	None		Edit Delete

[Home](#)

5.5.3 Add record

Before a supplier can upload products to the database, he must register himself on the database. In SQL terminology we need something like this:

```
INSERT INTO Supplier (supplier_name, address, email, phone_number)
VALUES ('Mike`s online shop', '12 OR Tambo drive, Bloemfontein, 9301',
'mike@mikeshop.co.za', '012 345 6789')
```

Reference:

- https://www.w3schools.com/django/django_add_record.php

5.5.3.1 Template

Under `..\templates\suppliers`, add a new empty file, `supplier_add.html`.

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

{% block scripts %}
{% endblock %}

{% block content %}

    <h2>ADD SUPPLIER</h2>

    <form action="/supplier_add_post/" method="post">
        {% csrf_token %}
        <table>
            <tr>
                <td>Supplier name</td>
                <td><input name="name"></td>
            </tr>
            <tr>
                <td>Address</td>
                <td><textarea rows="5" cols="60" name="address"></textarea></td>
            </tr>
            <tr>
                <td>Email address</td>
                <td><input name="email"></td>
            </tr>
            <tr>
                <td>Phone number</td>
                <td><input name="phone"></td>
            </tr>
        </table>
    </form>
```

```

        <tr>
            <td colspan="2" align="center">
                <p></p>
                <input type="submit" value="Add">
                <input type="button" value="Cancel"
                    onclick="window.location='/supplier_list/'">
                <p></p>
            </td>
        </tr>
    </table>
</form>
{% endblock %}

```

Notes

- User inputs are placed inside an html form.
- The form's method attribute indicates that the data entered by the user in the form's input controls will be returned (posted) back to the server where it will be handled.
- The form's action attribute indicates what will happen when the user clicks the Submit button. In this case, the view `supplier_add_post` will be called. The form inputs will be retrieved and handled.
- An `<input>` tag without indication of type is a text box in which the user can enter a value. The name attribute is important to identify the tag when the form is submitted.
- There is no input for the `supplier_id`. That is an auto-number field and is generated automatically by the database when the record is saved.
- The `<input type="submit">` tag is a button. When the user clicks the button, the form is closed and the values in the input tags are available for retrieval by the server. The value attribute determines the text that is shown in the button.
- The `<input type="button">` tag is a button where the behaviour is defined in the `onclick` attribute. In this case, the form will be closed and the `supplier_list` url is shown.
- `{% csrf_token %}` is a Django tag that provides protection against Cross Site Request Forgeries. This tag is compulsory in Django and you will get an error if it is omitted. Read about this here:

<https://docs.djangoproject.com/en/4.1/ref/csrf/>

<https://www.squarefree.com/securitytips/web-developers.html#CSRF>

5.5.3.2 Views

In `vw_suppliers.py`, add the following two views. The first view can be called by a user in a url. The second one is called by the form in the template after the user pressed the submit button.

```

def supplier_add(request):
    return render (request, "supplier_add.html")

def supplier_add_post(request):
    if request.method == 'POST':
        name = request.POST['name']
        address = request.POST['address']
        email = request.POST['email']
        phone = request.POST['phone']
        supplier = Supplier(supplier_name=name, address= address, email=email,\
                           phone_number = phone)
        supplier.save()
    return supplier_list(request)

```

Notes

- We first check if the method was called via a form with post action. If the user calls the view directly from the browser's url, this code will not be executed.
- `request.POST['name']`
 - A dictionary object containing all the values of the html elements in the template with the respective name. Make sure that the parameter is spelled exactly as in the template.
 - Note that POST must be all capitals.
- `return supplier_list(request)`
 - Returns to the list of suppliers

5.5.3.3 Paths

```
from app_clothing.views.vw_suppliers import supplier_list, supplier_add, supplier_add_post

urlpatterns = [
    ...
    #Suppliers
    path('supplier_list/', supplier_list),
    path('supplier_add/', supplier_add),
    path('supplier_add_post/', supplier_add_post),
]
```

Note:

- The forward slash after the urls.

5.5.3.4 Test

SUPPLIERS

ID	Name	Address	Telephone	
1	Supplier 1	None		Add supplier
2	Supplier 2	None		Edit Delete
3	Supplier 3	None		Edit Delete
5	Supplier 5	None		Edit Delete
8	Supplier 5	None		Edit Delete
4	Supplier4	None		Edit Delete

[Home](#)

ADD SUPPLIER

Supplier name

Address

Email address

Phone number

After you click Add, the new supplier should be listed.

5.5.4 Edit a record

A supplier might realise that he made a mistake when adding a record and needs a way to change (edit) existing data.

Reference:

- https://www.w3schools.com/django/django_update_record.php

5.5.4.1 Template

Under `..\templates\suppliers`, add a new empty file, `supplier_edit.html`.

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

{% block scripts %}
    As for supplier_add/html
{% endblock %}
```

```
{% block content %}

<h2>EDIT SUPPLIER</h2>

<form action="/supplier_edit_post/{{supplier.supplier_id}}/" method="post">
    {% csrf_token %}
    <table>
        <tr>
            <td>Supplier name</td>
            <td><input name="name" value="{{ supplier.supplier_name }}"></td>
        </tr>
        <tr>
            <td>Address</td>
            <td>
                <textarea rows="5" cols="60" id="taAddress"
                    onchange="UpdateHidden();">{{supplier.address}}</textarea>
                <input type="hidden" id="hiddenAddress" name="address"
                    value="{{supplier.address}}"></hidden>
            </td>
        </tr>
        ... Other rows
    </table>
    <tr>
        <td colspan="2" align="center">
            <p></p>
            <input type="submit" value="Submit">
            <input type="button" value="Cancel"
                onclick="window.location='/supplier_list/'">
            <p></p>
        </td>
    </tr>
</form>
{% endblock %}
```

Notes

- This form looks very similar to the supplier_add form. Note the differences:
 - The action method is different and takes a parameter with the primary key of the record that must be edited. This parameter will identify the record to edit in the view.
 - The value parameter of input elements is specified so that the user can see the current values.

5.5.4.2 Views

In `vw_suppliers.py`, add the following two views. The first view can be called by a user in a url. The second one is called by the form in the template after the user pressed Submit.

```
def supplier_edit(request, supplier_id):
    supplier = Supplier.objects.get(supplier_id = supplier_id)
    return render (request, "supplier_edit.html", {'supplier': supplier})

def supplier_edit_post(request, supplier_id):
    if request.method == 'POST':
        supplier = Supplier.objects.get(supplier_id=supplier_id)
        supplier.supplier_name = request.POST['name']
        supplier.address = request.POST['address']
        supplier.email = request.POST['email']
        supplier.phone_number = request.POST['phone']
        supplier.save()
    return supplier_list(request)
```

Notes

- The views take an extra parameter to identify the record that must be edited. This parameter is specified in the URL.
- The specific supplier is retrieved from the model `Supplier` through the `objects.get()` method and usage of the supplier's primary key.
- In `supplier_edit`, the context parameter is used to send supplier through to the template. The supplier's attributes can then be used to populate the various input tags.
- In `supplier_edit_post`, the supplier's attributes are assigned values that are retrieved from the template through the `POST` dictionary.
- The changes are saved to the database.
- The browser is returned to the list of suppliers.

5.5.4.3 Paths

```
from app_clothing.views.vw_suppliers import ..., supplier_edit, supplier_edit_post

urlpatterns = [
    ...,
    path('supplier_edit/<int:supplier_id>', supplier_edit),
    path('supplier_edit_post/<int:supplier_id>', supplier_edit_post),
]
```

5.5.4.4 Test



EDIT SUPPLIER

Supplier name

Address

Email address

Phone number

Click on Submit and check that the details was changed.

5.5.5 Delete a record

It should always be possible to delete a record from the database.

Reference:

- https://www.w3schools.com/django/django_delete_record.php

5.5.5.1 Template

We actually do not need a template for this, but it is good practice to provide the user with a confirmation. Under `..\templates\suppliers`, create a new file, `supplier_delete.html`.

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

{% block content %}

<h2>DELETE SUPPLIER</h2>
```

```

<form action="/supplier_delete_post/{{ supplier.supplier_id }}" method="post">
  {% csrf_token %}
  <p>Are you sure you want to delete {{ supplier.supplier_name }}?</p>
  <br>
  <div align="center">
    <input type="submit" class="submit-btn" value="&nbsp;Yes&nbsp;">
    <input type="button" class="submit-btn" value="&nbsp;No&nbsp;"
    onclick="window.location='/supplier_list/'">
    <p></p>
  </div>
</form>
{% endblock %}

```

5.5.5.2 View

In `vw_suppliers.py`, add the following two views.

```

def supplier_delete(request, supplier_id):
    supplier = Supplier.objects.get(supplier_id = supplier_id)
    return render (request, "supplier_delete.html", {'supplier': supplier})

def supplier_delete_post(request, supplier_id):
    supplier = Supplier.objects.get(supplier_id = supplier_id)
    if request.method == 'POST':
        supplier.delete()
    return supplier_list(request)

```

5.5.5.3 Paths

```

from app_clothing.views.vw_suppliers import ..., supplier_delete, supplier_delete_post

urlpatterns = [
    ...,
    path('supplier_delete/<int:supplier_id>', supplier_delete),
    path('supplier_delete_post/<int:supplier_id>', supplier_delete_post),
]

```

5.5.5.4 Test

Click on Delete for one of the suppliers:



Click on No and check that the supplier was not removed.
Click on Yes and check that the supplier was removed.

5.5.5.5 Make provision for errors

If your foreign keys are defined properly, you should not be able to delete a supplier if there is a related product for that supplier in the `Product` table.

To present a user with a message like this is unforgiveable. The poor user would not know what has hit him!



Add a message box as explained in the previous chapter. Then, change `supplier_delete_post`:

```
def supplier_delete_post(request, supplier_id):
    supplier = Supplier.objects.get(supplier_id = supplier_id)
    if request.method == 'POST':
        try:
            supplier.delete()
        except:
            return vw_home.show_message(request, \
                "Cannot delete supplier '" + supplier.supplier_name \
                + "'.\nA related product exists.")
    return supplier_list(request)
```



5.5.6 Exercise

In Section 5.3 above, we presented some basic examples of listing customers and products.

- Add templates and views for `customer_list`, `customer_add`, `customer_edit` and `customer_delete`. Remember to add the customers template subfolder in `settings.py`.
- Replace the `product_list` template and view to look like the ones for `supplier_list`. Add templates and views for `product_add`, `product_edit` and `product_delete`.

5.6 Select an item from a dropdown list

When we have a foreign key in a model, we would like to select an item from an available list instead of allowing the user to enter a value in a text box. The user might make a mistake and enter a value for which the referential integrity will be violated.

In the example below, we allow a user to add/edit a product and select the supplier of the product from a list.

5.6.1 Template

```
<tr>
  <td>Supplier</td>
  <td>
    <select name="supplier_id">
      {% for supplier in supplier_list %}
        {% if product.supplier_id == supplier.supplier_id %} #Select current value
          <option value="{{ supplier.supplier_id }}" selected>{{ supplier.name }} </option>
        {% else %}
          <option value="{{ supplier.supplier_id }}">{{ supplier.name }} </option>
        {% endif %}
      {% endfor %}
    </select>
  </td>
</tr>
```

```

        {% endif %}
    {% endfor %}
</select>
</td>
</tr>

```

Notes:

- The supplier name (option text) is displayed for the user but the supplier id (value attribute) is sent back to the server on submit.

5.6.2 View

We need to send the list of suppliers through to the template via the context argument:

```

def product_add(request):
    suppliers = Supplier.objects.all();
    return render(request, 'product_add.html', {"supplier_list": suppliers} )

```

5.7 User-defined filters

A user should have control over filtering. We can add a bar at the top of a page with a list of products to allow the user to select the supplier(s) for which the products must be displayed. We can use a session variable to remember the user's selection so that it will remain constant when he moves to another page and returns to the list of products.

PRODUCTS (3)				
Suppliers: <input type="checkbox"/> Supplier 1 <input type="checkbox"/> Supplier 2 <input type="checkbox"/> Supplier 3 <input checked="" type="checkbox"/> Supplier 4 <input checked="" type="checkbox"/> Supplier 5				
Code	Description	Price	Supplier	
a	Product A	100.00	Supplier 5	Add product Edit Delete
b	Product B	200.00	Supplier 4	Edit Delete
D	Product D	402.00	Supplier 4	Edit Delete

[Home](#)

5.7.1 Template

Remember that client and server are two separate entities and no direct interaction is possible. When the user changes a selection, the choice must be posted back to the server and the entire page has to be reloaded from scratch. Previously, the template for `product_list.html` did not contain a form. Now we need a form to allow the user to post changes.

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

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

```

<h2>PRODUCTS (3)</h2>

<form id="id_home" action="/product_list/" method="post" enctype="multipart/form-data">
    {% csrf_token %}

    <!-- Suppliers -->
    <div style="position: relative; display: flex;
        background-color: darkgoldenrod; color: white; font-weight: bold;">
        &nbsp;Suppliers: &nbsp;
        {% for supplier in suppliers %}
            <input type="checkbox" name="checked_suppliers" value="{{supplier.supplier_id}}"
                {% if supplier.supplier_id in selected_suppliers %}
                    checked
                {% endif %}
                onclick="document.forms[0].submit();">
                {{ supplier.supplier_name }}
            </input>
            &nbsp;
        {% endfor %}
    </div>

    <!-- Column headers -->
    ...

    <!-- Data rows -->
    ...

    <input type="button" value="Home" onclick="window.location='/'">
</p></p>

</form>

{% endblock %}

```

Notes

- The form's action attribute is set to return to the same view where it has been generated from.
- A series of checkboxes is displayed – one for each supplier in the context variable suppliers.
- The checkboxes all have the same name – checked_suppliers.
- A context variable, selected_suppliers, is a list of suppliers that have been selected by the user with a prior post.
- Check boxes in selected_suppliers are checked.
- When a check box is clicked (checked or unchecked), the form is submitted.

5.7.2 Views

In order not to lose the previous version of the views, I added new views with different names.

```

def product_list_3(request):
    #All suppliers
    suppliers = Supplier.objects.all()

    #Selected suppliers - check if the session variable is defined
    # We use a session variable to remember the last selected suppliers in case
    # the user goes to another screen and then returns here
    try:
        selected_suppliers = request.session['selected_suppliers']
    except:
        selected_suppliers = [] #Empty suppliers list
        #selected_suppliers = suppliers.values_list('supplier_id', flat=True) #All suppliers

    #Products of selected suppliers
    products = Product.objects.filter(supplier__supplier_id__in= selected_suppliers).order_by('description')

    #Check if the user changed the filter fields - overwrite selected suppliers
    if request.method=="POST" \

```

```

        and 'product_list' in request.path: #Check if the request did not perhaps come from another
                                            page, e.g. edit_post
            selected_suppliers_s = request.POST.getlist('checked_suppliers')
            selected_suppliers = [int(id) for id in selected_suppliers_s] #Map list of str to list of int
            request.session['selected_suppliers'] = selected_suppliers
            products = Product.objects.filter(supplier__supplier_id__in=selected_suppliers).order_by('description')
        #else request came from elsewhere - keep the initial products list

        #Render the html for the filled-in template
        context = {'suppliers': suppliers, 'selected_suppliers': selected_suppliers, 'products': products, }
        return render (request, "product_list_3.html", context)

def product_list_3b(request, supplier_id=-1):
    #Update the session variable for selected suppliers
    if supplier_id == -1:
        request.session['selected_suppliers'] = []
    else:
        request.session['selected_suppliers'] = [supplier_id]
    #Proceed to the standard view to filter the products and display them in a template
    return product_list_3(request)

def product_add_post(request):
    ...
    return product_list_3(request)

def product_edit_post(request, product_id):
    ...
    return product_list_3(request)

def product_delete_post(request, product_id):
    ...
    return product_list_3(request)

```

Notes

- The first view is called when no supplier is provided in a query param.
- Previously, the `product_list` view was not executed after a post. Now, the user could change the filters and we need to check for it. But: The post could also have come from another page, e.g. `product_add_post` or `product_edit_post`. So, we also need to check that the post was done by `product_list`.
- Note the comments in the code – it should suffice to explain the logic.
- The second view above is called when a specific supplier is provided, e.g. from the link of Products on the Suppliers page. This will overwrite the session variable with the single selected supplier and then proceed to the first view.
- All the other views where the return was specified to return to `product_list` had to be updated with the new view name.

5.7.3 Paths

```

from app_clothing.views.vw_products import product_list_3, product_list_3b

urlpatterns = [
    ...
    #Products
    path('product_list/', product_list_3),
    path('product_list/<int:supplier_id>/', product_list_3b),

```

Notes

- The first path is for the normal call with no query parameters, e.g. from the index.
- The second path is for a call with query params, e.g. from the Products link on the Suppliers page.

5.7.4 Other types of filters

The user should also have control over sort order, date ranges, price intervals, etc. For example, to allow the user control over sort order, we could update the above code like this:

Template

```
<form id="id_home" action="/product_list/" method="post" enctype="multipart/form-data">
  {% csrf_token %}

  <!-- Suppliers -->
  ...
  <!-- Order by -->
  <div style="position: relative; display: flex; margin-bottom: 20px;
    background-color: darkgoldenrod; color: white; font-weight: bold;">
    &nbsp;Order by: &nbsp;  
    <input type="radio" name="order_by" value="product_code"
      onclick="document.forms[0].submit();"
      {% if order_by == "product_code" %} checked {% endif %}>
      Product code&nbsp;  
    <input type="radio" name="order_by" value="supplier"
      onclick="document.forms[0].submit();"
      {% if order_by == "supplier" %} checked {% endif %}>
      Supplier, Product code&nbsp;  
    <input type="radio" name="order_by" value="price"
      onclick="document.forms[0].submit();"
      {% if order_by == "price" %} checked {% endif %}>
      Price&nbsp;  
  </div>

  <!-- Column headers -->
  ...
  <!-- Data rows -->
  ...
</form>
```

View

```
def product_list_3(request):
    ...
    #Check if the session vars changed
    try: selected_suppliers = request.session['selected_suppliers']
    except: selected_suppliers = [] #Empty suppliers list
    try: order_by = request.session['order_by'];
    except: order_by = 'product_code'

    #Products of selected suppliers
    products = Product.objects\
        .filter(supplier__supplier_id__in= selected_suppliers)\
        .order_by(order_by, 'product_code')

    #Check if the user changed the filter fields - overwrite selected suppliers
    if request.method=="POST" and 'product_list' in request.path:
        ...
        order_by = request.POST.get('order_by')
        request.session['order_by'] = order_by
        products = Product.objects \
            .filter(supplier__supplier_id__in=selected_suppliers)\
            .order_by(order_by, 'product_code')

    context = {'suppliers': suppliers, 'selected_suppliers': selected_suppliers,\
        'products': products, 'order_by' : order_by}
    return render (request, "product_list_3.html", context)
```

PRODUCTS (3)				
Suppliers: <input checked="" type="checkbox"/> Supplier 1 <input type="checkbox"/> Supplier 2 <input type="checkbox"/> Supplier 3 <input checked="" type="checkbox"/> Supplier 4 <input checked="" type="checkbox"/> Supplier 5				
Order by: <input checked="" type="radio"/> Product code <input type="radio"/> Supplier, Product code <input type="radio"/> Price				
Code	Description	Price	Supplier	Add product
a	Product A	100.00	Supplier 5	Edit Delete
b	Product B	200.00	Supplier 4	Edit Delete
D	Product D	402.00	Supplier 4	Edit Delete
E	Product E	500.00	Supplier 1	Edit Delete
Home				

5.8 Summary

In this chapter, we focused on client access to data. We kicked off with a basic understanding of what QuerySets are. This is crucial to our understanding of how data is selected, filtered and presented. We provided basic examples of how to retrieve a single object or a list of objects. Next, we presented more advanced examples of creating, updating and deleting objects of a class. Finally, we showed how to allow the user control over filters and the sequence of objects.

The next chapter will focus on user administration.

Chapter 6: User administration

6.1 Built-in user administration

Django is a framework that automates many things, including user administration for a site. This means that you do not have to do this yourself when developing a new site.

References

- <https://docs.djangoproject.com/en/4.1/#the-admin>
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Admin_site
- <https://docs.djangoproject.com/en/4.1/topics/auth/default/#user-objects>

6.1.1 Admin user

Make sure that the password validations in `settings.py` is all commented out. You must uncomment them again when you publish the website.

In a terminal:

```
..\pr_clothing> python manage.py createsuperuser
```

Follow the prompts. Don't forget the password that you entered. For debugging you can enter a simple and short username/password, e.g. `clothing/clothing`.

Note:

- There is no feedback when you type the password. This is normal.

6.1.2 Register models for admin

Under `..\pr_clothing\app_clothing`, find `admin.py` and edit as follows. This will register your models for the admin user and allow him/her to add, edit and remove objects.

```
from django.contrib import admin

# Register your models here.

from .models import Customer, Supplier, Product, Sale
admin.site.register(Customer)
admin.site.register(Supplier)
admin.site.register(Product)
admin.site.register(Sale)
```

Save.

Warning:

- This might create a circular import error if you use `inspectdb` to update the models from the database. If you want to use `inspectdb`, temporarily comment out the register lines in the code above.

6.1.3 Use the admin site

Once the admin user is defined, you can enter this in a browser and then login with the admin account:

localhost:8000/admin

- Use the provided interface to add, edit, remove customers, suppliers, products, and sales records.

If you want to provide the user with a link to access the admin site, add an entry in `index.html` and in `urls.py`:

`index.html`

```
<ul>
...
<li><a href="/admin/" target="_blank">Administration</a></li>
...
</ul>
```

`urls.py`

```
from django.contrib import admin

urlpatterns = [
    path('admin/', admin.site.urls),
    ...
]
```

6.2 Authentication

<https://docs.djangoproject.com/en/4.1/topics/auth/default/#authenticating-users>
<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication>

6.2.1 Authentication vs authorisation

Authentication refers to the process to determine the identity of a user. It is normally done by requesting a user to log in with a registered username and password. Thereafter, the user can be allowed or denied certain functionalities depending on their status.

- **Authentication** is the process of confirming if a user has access to a system.
- **Authorisation** pertains to what the "authenticated" user can do in a system.

Put another way, authentication answers the question 'who are you?' while authorization answers 'what can you do?'. (<https://testdriven.io/blog/django-permissions/>)

6.2.2 Built-in forms

Django contains a built-in user management system. If we import from `django.contrib.auth` import views as `auth_views`, we can use the built-in forms for login, log out, password change, etc.

<https://docs.djangoproject.com/en/4.1/topics/auth/default/#module-django.contrib.auth.forms>

<https://docs.djangoproject.com/en/4.1/topics/auth/default/#module-django.contrib.auth.views>

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

6.2.2.1 Include in `./pr/pr/urls.py`

<https://docs.djangoproject.com/en/4.2/topics/auth/default/#using-the-views>

```
from django.urls import include, path

urlpatterns = [
    ...,
    path("accounts/", include("django.contrib.auth.urls")),
]
```

Now, the following URLs are available:

```
.../accounts/login
.../accounts/password_change
.../accounts/password_reset
```

6.2.2.2 Log in

<https://learndjango.com/tutorials/django-login-and-logout-tutorial>

<https://docs.djangoproject.com/en/4.2/topics/auth/default/#django.contrib.auth.views.LoginView>

View

- From the Django docs: "It's your responsibility to provide the html for the login template, called `registration/login.html` by default." So, under templates, create a new folder, `registration`.
- If you prefer to name the `login.html` differently or save it elsewhere, you specify the `template_name` parameter in `urls.py`:

```
path("accounts/login/", auth_views.LoginView.as_view(template_name="myapp/login.html")),
```
- A minimal login template can look like this:

```
<h2>Log In (built-in form)</h2>
<form method="post">
```

```
{% csrf_token %}
{{ form.as_p }}
<button type="submit">Log In</button>
</form>
```

- The `form` variable is passed as context from `auth.views.LoginView` and refers to the authentication form that contains the username and password input elements.
- See this for a more advanced login html:
<https://docs.djangoproject.com/en/4.1/topics/auth/default/#django.contrib.auth.views.LoginView>
- Tell your app where to go to after successful login. Under `../pr/pr/settings.py`, redirect to the home page after login:

```
LOGIN_REDIRECT_URL = '/'
```

- If you want to call the login page from index:
`Log in`

6.2.2.3 Change or reset password

There is no need for templates as for login unless you want to customise them and not use the Django administration forms.

- `../accounts/password_change`
- `../accounts/password_reset`

The image displays two side-by-side screenshots of Django's built-in password management forms. The left screenshot is the 'Password change' form, which includes a header 'Django administration', a breadcrumb 'Home > Password change', and a title 'Password change'. The instructions state: 'Please enter your old password, for security's sake, and then enter your new password twice so we can verify you typed it in correctly.' It features three input fields: 'Old password:', 'New password:', and 'New password confirmation:'. At the bottom is a blue button labeled 'CHANGE MY PASSWORD'. The right screenshot is the 'Password reset' form, also with the 'Django administration' header and a breadcrumb 'Home > Password reset', titled 'Password reset'. The instructions say: 'Forgotten your password? Enter your email address below, and we'll email instructions for setting a new one.' It has a single 'Email address:' input field and a blue button labeled 'Reset my password'.

6.2.3 Custom log in/out

For me it is easier, and I feel that I have more control if I do not use the built-in authentication forms.

6.2.3.1 Template

Under `..\templates`, create a folder, `users`. In this folder, add a file, `user_login.html`:

```
{% extends "base.html" %}
{% block content %}
```

```

<form action="/user_login_post/" method="post">
    {% csrf_token %}

    <table>
        <tr>
            <td width="100%" colspan="2" align="center">
                <h1>Log in (custom)</h1>
            </td>
        </tr>
        <tr>
            <td>User name</td>
            <td><input name="username"></td>
        </tr>
        <tr>
            <td>Password</td>
            <td><input type="password" name="password"></td>
        </tr>
        <tr><td><br></td></tr>
        <tr>
            <td colspan="2" align="center">
                <input type="submit" value="Log in">
            </td>
        </tr>
    </table>
</form>
{% endblock %}

```

Notes

- The form's action attribute indicates what will happen when the user clicks the Submit button. In this case, the view user_login_post will be called. The form inputs will be retrieved and handled.
- The password input must be of type="password".

6.2.3.2 Views

- Under views, create a file, vw_users.
- Import the necessary modules.
- Add a view to show the custom user_login.html template.
- Add a user_login_post view which will be called from the user_login template when the user clicks on Submit.
- Add also a user_logout view which will be called from the browser when the user wants to logout of the system.

```
from django.contrib.auth import authenticate, login, logout
```

```

def user_login(request):
    return render(request, 'user_login.html')

def user_login_post(request):
    if request.method == 'POST':
        un = request.POST['username']
        pwd = request.POST['password']
        user = authenticate(username=un, password=pwd)
        if user is not None:
            login(request, user)
        else:
            logout(request)
            messages.error(request, "Authentication failed")
    else:
        messages.error(request, "Not posted")
    return vw_home.index(request)

def user_logout(request):

```

```
logout(request)
return vw_home.index(request)
```

Notes:

- In the `user_login_post` method:
 - Authenticate the user and login to the system if authenticated.
 - If the user entered the wrong credentials, logout the current user.
 - We used the `messages` facility of Django to give feedback to the user.
 - Note the way in which a user is authenticated.

6.2.3.3 Paths

Add the necessary url patterns for the views added above.

```
from app_clothing.views.vw_users import user_login, user_login_post, user_logout

urlpatterns = [
    ...,
    path('user_login/', user_login),
    path('user_login_post/', user_login_post),
    path('user_logout/', user_logout),
]
```

6.2.3.4 Adapt the index

Change the index file to allow the user to log in and out. Only the menu options are displayed for which the user is authorised.

```
{% extends "base.html" %}
{% block content %}
    <h2>Clothing shop</h2>
    <ul>
        <li><a href="/user_login/">Log in</a></li>
        {% if request.user.is_authenticated %}
            <p>User name: {{ request.user.username }}</p>
            <li><a href="/customers/">Customers (raw)</a></li>
            <li><a href="/customer_list/">Customers</a></li>
            <li><a href="/supplier_list/">Suppliers</a></li>
            <li><a href="/product_list/">Products</a></li>
            {% if request.user.is_superuser %}
                <br>
                <li><a href="/user_list/">Users</a></li>
                <li><a href="/admin/"
                    target="_blank">Administration</a></li>
                <br>
                <li><a href="/SQL/">SQL</a></li>
            {% endif %}
            <li><a href="/user_logout/">Log out</a></li>
        {% endif %}
    </ul>
{% endblock %}
```

Clothing shop

- [Log in](#)
- User name: a
- [Customers \(raw\)](#)
- [Customers](#)
- [Suppliers](#)
- [Products](#)
- [Users](#)
- [Administration](#)
- [SQL](#)
- [Log out](#)

Notes

- Note the syntax to authenticate and authorise a user.
(<https://docs.djangoproject.com/en/4.1/topics/auth/default/#users>)
- Note how `request.user` is used to access the currently logged in user.
(<https://docs.djangoproject.com/en/4.2/ref/contrib/auth/#attributes>)

6.3 User operations

6.3.1 List users

There are 3 options to get a list of registered user on the site:

1. SQL query: `SELECT * FROM auth_user;`
2. Admin site
3. Programmatically. See the details below.

6.3.1.1 Template

```
{% extends "base.html" %}
{% block content %}

<h1>Users</h1>

<!-- Column headers -->
<table>
  <tr>
    <th width="70px" align="left">User id</th>
    <th width="120px" align="left">User name</th>
    <td></td>
  </tr>
</table>

<!-- Data rows -->
<table>
  {% for user in user_list %}
    <tr>
      <td width="70px">{{ user.id }}</td>
      <td width="120px">{{ user.username }}</td>
      <td><a href="/user_password/{{ user.id }}">Change password</a></td>
    </tr>
  {% endfor %}
</table>

<div align="center">
  <br>
  <input type="button" value="Home" onclick="window.location='/'">
</div>

{% endblock %}
```

Users		
User id	User name	
1	clothing	Change password
2	a	Change password
3	c	Change password
4	s	Change password
5	p	Change password
<input type="button" value="Home"/>		

Notes

- We do not have links here to add or remove users because that should be done through the built-in admin site. (Unless we have implicit user tables.)
- The link to change a password will be explained in the next section.

6.3.1.2 View

```
def user_list(request):
    users = User.objects.all()
    return render (request, "user_list.html", {'user_list': users})
```

6.3.1.3 Path

```
from app_clothing.views.vw_users import user_list, ...

urlpatterns = [
    path('user_list/', user_list),
    ...
]
```

6.3.2 Change password

<https://docs.djangoproject.com/en/4.1/topics/auth/default/#changing-passwords>

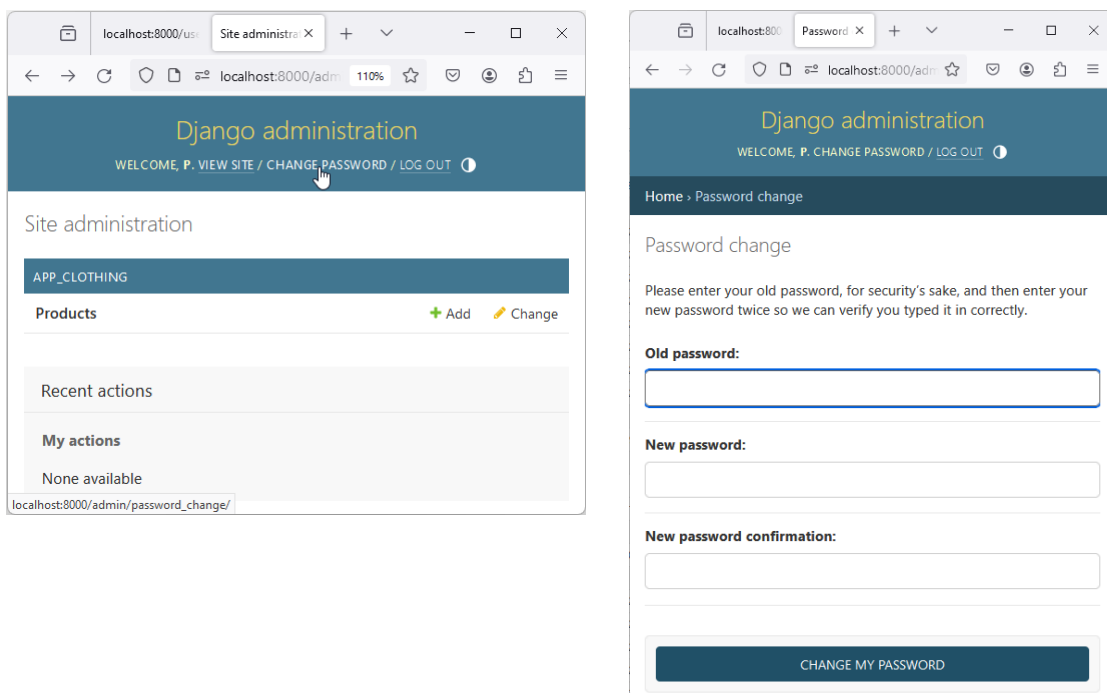
1. In the console:
~/.pr_clothing> python manage.py changepassword <username>
2. Through the admin site
 - Click on Users, select the user, under "Password" click on "this form".
3. Through the built-in forms (next section):
 - http://.../accounts/password_change/
4. Programmatically

```
user = User.objects.get(id = user_id)
user.set_password(new_password)
user.save()
```

A user must be allowed to change his own password. This can be done through the admin site or programmatically.

6.3.2.1 Through the admin site

The user p in our set of test users is a normal user with staff access only. For him, the built-in site would look like this.



6.3.2.2 Programmatically

Template

```
{% extends "base.html" %}
{% block content %}

<h1>Change user password</h1>

<form action="/user_password_post/{{ user.id }}"
```

Change user password

Username

a

Old password

New password

New password again

Submit

Cancel

```

        method="post">
{% csrf_token %}
<table>
    <tr>
        <td>Username</td>
        <td>{{ user.username }}</td>
    </tr>
    <tr>
        <td>Old password</td>
        <td><input type="password" name = "old_password"></td>
    </tr>
    <tr>
        <td>New password</td>
        <td><input type="password" name="new_password"></td>
    </tr>
    <tr>
        <td>New password again</td>
        <td><input type="password" name="new_password_again"></td>
    </tr>

    <tr></tr>
    <tr>
        <td></td>
        <td>
            <input type="submit" value="Submit">
            <input type="button" value="Cancel" onclick="window.location='/user_list/'">
        </td>
    </tr>
</table>
</form>
{% endblock %}

```

Notes

- The password inputs must be of type="password" to mask the user entries.
- It is good practice to ask for the old password and to enter the new password twice.

Views

```

def user_password(request, user_id):
    user = User.objects.get(id=user_id)
    return render (request, "user_password.html", {'user': user})

def user_password_post(request, user_id):
    if request.method == 'POST':
        user = User.objects.get(id = user_id)
        old_password = request.POST['old_password'];
        new_password_1 = request.POST['new_password'];
        new_password_2 = request.POST['new_password_again'];
        if user.check_password(old_password) \
            and new_password_1 == new_password_2:
            user.set_password(new_password_1);
            user.save()
            return vw_home.show_message(request, "Password changed")
        else:
            return vw_home.show_message(request, "Invalid old password or new passwords do not match.")
    return user_list(request)

```

Notes:

- The password is hashed and we cannot do


```
if user.password == (old_password):
```

 The check_password method hashes the given password and then compares it with the existing user password.

Paths


```

from app_clothing.views.vw_users import user_list, user_login, user_login_post, \
    user_logout, user_password, user_password_post

urlpatterns = [
    ...
    #Users
    path('user_list/', user_list),
    path('user_login/', user_login),
    path('user_logout/', user_logout),
    path('user_login_post/', user_login_post),
    path('user_password/<int:user_id>/', user_password),
    path('user_password_post/<int:user_id>/', user_password_post),
] #end urlpatterns

```

6.3.3 Add user

Three options:

1. SQL:
`INSERT INTO auth_user (username, password) VALUES('john', 'glass onion');`
This is bad. It does not handle password hashing and should not be done.
2. Through the admin site
3. Programmatically

<https://stackoverflow.com/questions/10372877/how-to-create-a-user-in-django>

```

from django.contrib.auth.models import User
user = User.objects.create_user(
    username='john',
    email = 'jlennon@beatles.com',
    password='glass onion')

```

The above code just shows the basic syntax. Of course, you should use a template to allow the admin user to enter the details and then use `request.POST` to retrieve the entries. In the template, use the input element with `type="password"` for the password field.

Of the above options, using the admin site is by far the best. Direct access through sql does not allow for password hashing and doing it yourself through a template and views is error prone. There are some cases where we need to add a user programmatically as explained in the following section.

6.3.4 Implicit addition of new users

Depending on the scenario, it might happen that one or more entities in the database should also fulfil the role of a user.

In the running example of this tutorial, customers and suppliers must be registered as users. There is a 1:1 relationship between `User:Supplier` and `User:Customer`. So, we must add `auth_user` as a foreign table to both `Customer` and `Supplier`.

6.3.4.1 Update the models

- In `admin.py`, comment out all lines of code.
- In the console, pipe the changes to `models.py`.

- ```
..\env_clothing\pr_clothing> python manage.py inspectdb > app_clothing/models.py
```
- In models.py, check that the structures of Customer and Supplier was updated to show OneToOneField for user.

```
class Customer(models.Model):
 customer_id = models.AutoField(primary_key=True)
 user = models.OneToOneField('AuthUser', models.DO_NOTHING)
```

- In admin.py, uncomment all lines of code.

#### 6.3.4.2 Template

Change the previous template to add a supplier, to look like this. We added fields for the username and password.

```
<tr>
 <td>Username</td>
 <td><input name="username" onkeypress="return event.charCode != 32"></td>
</tr>
<tr>
 <td>Password</td>
 <td><input type="password" name="password" onkeypress="return event.charCode != 32"></td>
</tr>
```

Notes:

- onkeypress:  
Django does not allow spaces for the username and password. So, we must prevent the user from typing spaces.

#### 6.3.4.3 Views

The supplier\_add view stays the same as before. We have to change the supplier\_add\_post view to do a few more things:

- Whenever a new supplier is registered, we first have to register a new user and then use the new user's id as foreign key for the new supplier.
- Add the new user to the group of Suppliers. That will ensure that the new user has the correct permissions.  
(<https://stackoverflow.com/questions/6288661/adding-a-user-to-a-group-in-django>)
- Since we removed the email field from the Supplier table, we need to assign the user's entry for email to the email field of the new user.

- Create a Supplier object with the supplier id being the user id of the user that was created.

To do the above, we need to import the necessary libraries:

```
from django.contrib.auth.models import Group, Permission, User
```

Change the view for supplier\_add\_post as follows:

```
def supplier_add_post(request):
 if request.method == 'POST':
 #Get values from the template
 username = request.POST['username']
 password = request.POST['password']
 name = request.POST['name']
 address = request.POST['address']
 email = request.POST['email']
 phone = request.POST['phone']

 #Create new user and add the user to the appropriate group
 user = User.objects.create_user(username=username, password=password, email=email)
 group = Group.objects.get(name="Suppliers")
 group.user_set.add(user)

 #Create new upplier
 supplier = Supplier(supplier_name=name, user_id=user.id, \
 address= address, phone_number = phone)
 supplier.save()
 return supplier_list(request)
```

Since we removed the email field from the Supplier table, we have to update supplier\_edit\_post as follows. The supplier.user must be saved separately.

```
def supplier_edit_post(request, supplier_id):
 if request.method == 'POST':
 supplier = Supplier.objects.get(supplier_id=supplier_id)
 supplier.supplier_name = request.POST['name']
 supplier.address = request.POST['address']
 supplier.phone_number = request.POST['phone']
 supplier.save()
 supplier.user.email = request.POST['email']
 supplier.user.save();
 return supplier_list(request)
```

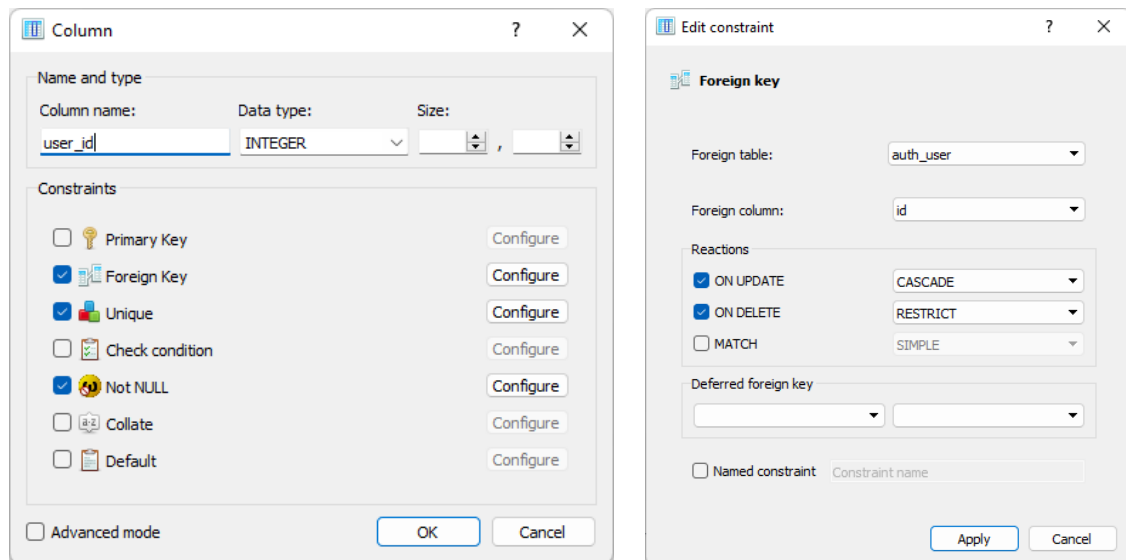
Notes

- Add a new supplier and check both the user and supplier lists that it has been added.
- Repeat the above procedures for the Customer table.

#### 6.3.4.4 Change the structure of existing tables

Use SQLite Studio:

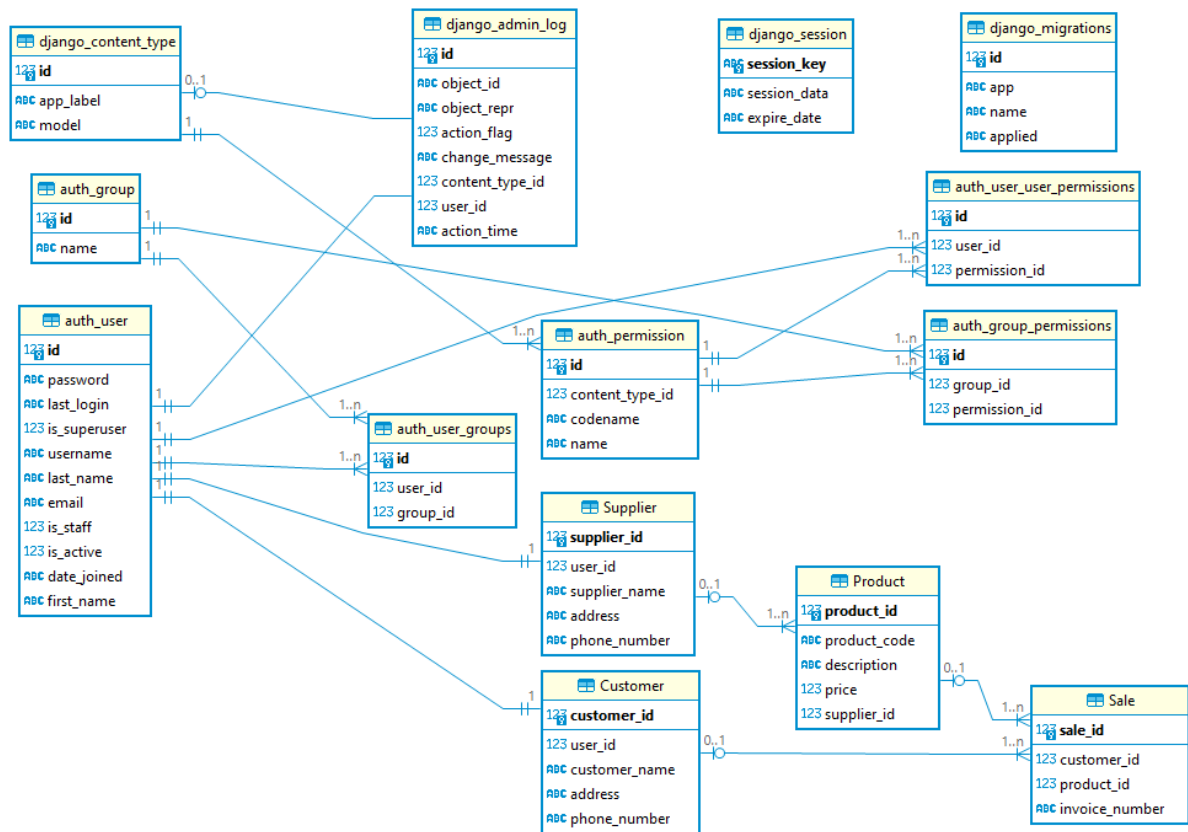
- Add a field, user\_id, to Customer. and configure it as a foreign key for auth\_user.



- We have to mark the `user_id` field as Not NULL as well, but that will fail since there are existing customers in the database for which the `user_id` field is empty.
- Remove all test data from the database and then mark the `user_id` field as Not NULL. Also mark it as Unique.
- Commit the changes and check that they were successful.
- Open the ERD in DBeaver. The relationship between `auth_user` and `Customer` should be automatically added. Check that it is a 1:1 relationship.
- You will notice that there is some duplication of fields in `auth_user` and `Customer`, for example both contain an `email` field. To preserve integrity and prevent confusion, we should remove the `email` field from `Customer`.
- Follow the same procedure for the `Supplier` table.

### 6.3.5 ERD of admin and custom tables

The ERD below shows the structure of the admin tables and how the `auth_user` table is connected with implicit user tables.



### 6.3.6 Remove user

#### 1. SQL:

```
DELETE FROM auth_user WHERE id = 13;
```

This expects you to know the id and handle confirmation yourself. This might fail if there are foreign key restrictions, for example related entries in `auth_user_groups`.

#### 2. Through the admin site

#### 3. Programmatically

```
user = User.objects.get(id = uID)
user.delete()
```

`user.delete()` takes care of all foreign key constraints in the background. So, related entries in `auth_user_groups` will be deleted before the entry in `auth_user` will be deleted.

The best option is of course through the built-in admin site, unless we have take care of the implicit removals as explained in the next section.

### 6.3.7 Implicit removal of users

There is a one:one relationship between `User:Customer` and `User:Supplier`. If we want to remove a customer, we should remove the corresponding user as well.

```
def supplier_delete_post(request, supplier_id):
 supplier = Supplier.objects.get(supplier_id = supplier_id)
 if request.method == 'POST':
 try:
 user_id = supplier.user.id
 supplier.delete()
```

```

 user = User.objects.get(id = user_id)
 user.delete()
 except:
 return vw_home.show_message(request, "Cannot delete supplier '" \
 + supplier.supplier_name \
 + "'.\nA related product exists.")

 return supplier_list(request)

```

Notes

- We obtain the user id to be deleted from the supplier object before it is deleted.

## 6.4 Permissions

This section might be challenging, but understanding is crucial to develop secure websites.

"Django comes with a built-in permissions system. It provides a way to assign permissions to specific users and groups of users."

<https://docs.djangoproject.com/en/4.1/topics/auth/default/#permissions-and-authorization>  
<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication>  
<https://testdriven.io/blog/django-permissions/>

Whenever a model is created, some default permissions are added with it. They can be viewed in the Django administration facility. For example, for the Supplier model, default permissions are "Can add supplier", "Can change supplier", "Can delete supplier", "Can view supplier".

### 6.4.1 Create groups and users and assign rights

- Use the built-in Django user administration site to create test groups and assign rights as in the table below.

Groups	Permissions
Customers	View and change Customer object
Suppliers	View and change Supplier object

- Add test users and assign them to groups as in the table below.

User name	Password	Groups
c11	c11	Customers
s11	s11	Suppliers

- Add test users and assign individual rights to them as in the table below.

User name	Password	Permissions
a	a	superuser
p	p	Do all with products

- Notes

- The usernames and passwords are very short to facilitate easier debugging. This can, of course, never be like this in a production environment.
- Note that we will either have to write code ourselves (which we will do later) or use another package, such as Django-guardian or Django-rules, to enforce stricter policies, for example to limit a customer to only edit his own profile and not that of other customers or limit a supplier to only edit products of which he is the supplier.

- Log in and out as different users and check that the rights are correctly applied.
- If you open the table `auth_user` from SQLite Studio, you should see the list of registered users.

## 6.4.2 Available and assigned permissions

Refer to the ERD above. The SQL queries below can be very handy to inspect the available permissions per model and the assigned permissions per group or user.

### 6.4.2.1 Available permissions per model

The `django_content_type` table contains a list of models for the app. If this table is joined with the `auth_permission` table, we can get a list of permissions that are available for each model.

```
SELECT model, codename, name
FROM django_content_type CT
 INNER JOIN auth_permission AP ON CT.id = AP.content_type_id
WHERE model in ('customer', 'supplier', 'product', 'sale')
ORDER BY model, codename
```

model	codename	name
customer	add_customer	Can add customer
customer	change_customer	Can change customer
customer	delete_customer	Can delete customer
customer	view_customer	Can view customer
product	add_product	Can add product
product	change_product	Can change product
product	delete_product	Can delete product
product	view_product	Can view product
sale	add_sale	Can add sale
sale	change_sale	Can change sale
sale	delete_sale	Can delete sale
sale	view_sale	Can view sale
supplier	add_supplier	Can add supplier
supplier	change_supplier	Can change supplier
supplier	delete_supplier	Can delete supplier
supplier	view_supplier	Can view supplier

### 6.4.2.2 Assigned users per group

```
SELECT username, G.name as `group`
FROM auth_group G
 INNER JOIN auth_user_groups UG ON G.id = UG.group_id
 INNER JOIN auth_user U ON U.id = UG.user_id
ORDER BY username, `group`
```

username	group
c	Customers
c1	Customers
c2	Customers
s	Suppliers
s1	Suppliers
s2	Suppliers
s3	Suppliers
s4	Suppliers
s5	Suppliers
s6	Suppliers
s7	Suppliers

#### 6.4.2.3 Assigned permissions per group

```
SELECT G.name AS group_name, codename, P.name AS permission_name
FROM auth_group G
 INNER JOIN auth_group_permissions GP ON G.id = GP.group_id
 INNER JOIN auth_permission P ON P.id = GP.permission_id
```

group_name	codename	permission_name
Customers	view_customer	Can view customer
Customers	change_customer	Can change customer
Suppliers	change_supplier	Can change supplier
Suppliers	view_supplier	Can view supplier

#### 6.4.2.4 Assigned permissions per user

```
SELECT U.id as user_id, username, codename, name
FROM auth_user U
 INNER JOIN auth_user_user_permissions UUP ON U.id = UUP.user_id
 INNER JOIN auth_permission P ON P.id = UUP.permission_id
```

user_id	username	codename	name
5	p	add_product	Can add product
5	p	change_product	Can change product
5	p	delete_product	Can delete product
5	p	view_product	Can view product
17	s7	change_supplier	Can change supplier

Remember, that if a user is part of a group, that group's permissions apply to the user and it is not necessary to assign the permission explicitly to individual users.

#### 6.4.3 Add permissions to a group

This is better done in the built-in admin site than programmatically.

#### 6.4.4 Add permissions to user

Permissions that are not part of the permissions of a group to which a user belong, can be assigned to individual users. It is not necessary to assign specific permissions to a user that belongs to a group who has that permission. It can be done in the built-in admin site or programmatically.

##### Programmatically

```
permission = Permission.objects.filter(codename='add_product').first()
if permission:
 user = request.user
 user.user_permissions.add(permission)
 user.save()
```

#### 6.4.5 Check permissions

	In view	In template
User type	<code>if request.user.is_authenticated:</code>	<code>{% if request.user.is_authenticated %}</code>



	<code>if request.user.is_superuser:</code>	<code>{% if request.user .is_superuser %}</code>
Group	<code>if request.user.groups \ .filter(name="Suppliers").exists():</code>	Need custom template filter. See the section below.
Permission	<code>if request.user \ .has_perm('app_clothing.add_product'):</code>	<code>{% if perms.app_clothing .change_supplier %}</code>

#### 6.4.5.1 Check if a user is in a group: View

Suppliers should not see the details of other suppliers. So, when the Suppliers url is loaded, we should filter the list:

```
def supplier_list(request):
 supplier_set = []
 if request.user.is_superuser:
 supplier_set = Supplier.objects.order_by('supplier_name')
 if request.user.groups.filter(name="Suppliers").exists():
 supplier_set = Supplier.objects.filter(user__id=request.user.id)
 context = {'supplier_set': supplier_set, }
 return render (request, "supplier_list.html", context)
```

#### 6.4.5.2 Check if a user is in a group: Template

We need to create a custom template filter.

- Under ../app\_, create a folder, templatetags.
- Under ../app\_/templatetags, create two files
  - \_\_init\_\_.py  
This is an empty file and indicates that all files in this folder are treated as Python packages.
  - has\_group.py

```
from django import template
register = template.Library()

@register.filter(name='has_group')
def has_group(user, group_name):
 return user.groups.filter(name=group_name).exists()
```

- In the template:
  - Once at the top: `{% load has_group %}`
  - Then where needed:  
`{% if request.user|has_group:"Suppliers" %}`
- If it does not work, restart the server.
- References
  - <https://docs.djangoproject.com/en/dev/howto/custom-template-tags/>
  - <https://stackoverflow.com/questions/34571880/how-to-check-in-template-if-user-belongs-to-a-group>

#### 6.4.5.3 Check if a user has a permission: View

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

if request.user.has_perm('app_clothing.add_product'):
 DoSomething
```

Notes

- app\_clothing above is the app\_label which is mostly the name of the app.
- has\_perm will always return true for superusers

#### 6.4.5.4 Check if a user has a permission: Template

The currently logged-in user's permissions are stored in the template variable `{{ perms }}`.

[https://stackoverflow.com/questions/9469590/check-permission-inside-a-template-in-](https://stackoverflow.com/questions/9469590/check-permission-inside-a-template-in-django)

[django](https://stackoverflow.com/questions/9469590/check-permission-inside-a-template-in-django)

<https://docs.djangoproject.com/en/4.1/topics/auth/default/#permissions>

You should change the template, `supplier_list.html`, so that suppliers can only see a link to edit their details. They should not be able to see the Add or Delete links.

```
{% if perms.app_clothing.change_supplier %}
 <td>Edit</td>
{% endif %}
```

- NB: See the "Be careful" section below. The fact that a user cannot see a link does not mean that he cannot type it directly in the url and execute it.

#### 6.4.6 Be careful

It is normally not a good idea to apply security policies on template level. A user can inspect the page source for guidance and type a url directly in the browser which would give him access to illegal facilities. The if statement above would make the link to add a customer invisible on the rendered page but nothing would prevent the user from typing the url directly into the browser. It is always better to apply security policies in the views before a page is rendered on the client.

- Only superusers should be able to add a supplier:

```
def supplier_add(request):
 if request.user.is_superuser:
 return render (request, "supplier_add.html")
 return vw_home.show_message(request, "Not permitted");
```

- Only superusers or the same supplier should be able to edit details:

```
def supplier_edit(request, supplier_id):
 supplier = Supplier.objects.get(supplier_id = supplier_id)
 if request.user.is_superuser or request.user.id == supplier.user.id:
 return render (request, "supplier_edit.html", {'supplier': supplier})
 return vw_home.show_message(request, "Not permitted")
```

- Only superusers should be able to delete a supplier:

```
def supplier_delete(request, supplier_id):
 if request.user.is_superuser:
 supplier = Supplier.objects.get(supplier_id = supplier_id)
 return render (request, "supplier_delete.html", {'supplier': supplier})
 return vw_home.show_message(request, "Not permitted")
```

You should update all GET views in the program accordingly.

Code that checks `request.METHOD == 'POST'` is normally not an issue because the user cannot execute that without having submitted a form – a form which he could not see if the GET view was properly protected.

Hint:

- It might be a good idea to leave all the links in the templates until such time that you tested all possibilities and ensured that you get a "Not permitted" message for all illegal possibilities. Only then can you start to hide the illegal possibilities – they only serve to not create false expectations from the users.

### 6.4.7 Object-level permissions

A supplier should only be able to add, edit or delete products for which he is the supplier. Customers should be able to see all products but they should not be able to add, edit or delete products. Superusers can add, edit or delete any product.

```
def product_add(request):
 suppliers = Supplier.objects.all();

 if request.user.is_superuser:
 return render(request, 'product_add.html', {"supplier_list": suppliers})

 if request.user.groups.filter(name="Suppliers").exists(): #Current user is a supplier
 supplier = Supplier.objects.get(user__id = request.user.id)
 return render(request, 'product_add.html', {'supplier': supplier})

 return vw_home.show_message(request, "Not permitted")

def product_add_post(request):
 if request.method == 'POST':
 pCode = request.POST['product_code']
 description = request.POST['product_description']
 price = request.POST['product_price']
 if request.user.is_superuser:
 supplier_id = request.POST['supplier_id']
 else:
 supplier_id = Supplier.objects.get(user__id = request.user.id).supplier_id
 product = Product(product_code=pCode, description=description, price=price, \
 supplier_id=supplier_id)
 product.save()
 return product_list_3(request)

def product_edit(request, product_id):
 product = Product.objects.get(product_id = product_id)

 if request.user.is_superuser:
 suppliers = Supplier.objects.all();
 return render(request, 'product_edit.html', {"product": product, \
 "supplier_list": suppliers})

 if request.user.id == product.supplier.user.id:
 supplier = product.supplier
 return render(request, 'product_edit.html', {"product": product, \
 "supplier": supplier})

 return vw_home.show_message(request, "Not permitted")

def product_edit_post(request, product_id):
 if request.method == 'POST':
 product = Product.objects.get(product_id=product_id)
 product.product_code = request.POST['product_code']
 product.description = request.POST['product_description']
 product.price = request.POST['product_price']
 if request.user.is_superuser:
 product.supplier_id = request.POST['supplier_id']
 else: pass #Supplier cannot change
 product.save()
 return product_list_3(request)

def product_delete(request, product_id):
```

```

product = Product.objects.get(product_id = product_id)
if request.user.is_superuser or request.user.id == product.supplier.user.id:
 product = Product.objects.get(product_id = product_id)
 return render(request, 'product_delete.html', {"product": product})
return vw_home.show_message(request, "Not permitted")

def product_delete_post(request, product_id):
 product = Product.objects.get(product_id = product_id)
 if request.method == 'POST':
 try:
 product.delete()
 except IntegrityError as e:
 messages.error(request, e.__cause__)
 return product_list_3(request)

```

## 6.5 Summary

In this chapter, we showed how to authenticate users, add and remove users, create and assign permissions and how to check if a user is authorised to do something on a site. Admin users should as far as possible use the built-in admin facility of Django to manage users, groups, and permissions. Security policies should be applied on the server before a page is rendered on the client.

# Chapter 7: Advanced features

## 7.1 Middleware

### 7.1.1 Introduction

Middleware defines functions that are executed in the background before and after a view is rendered to the browser. There are standard, built-in, default middleware functions and classes but we can also define our own custom middleware functions and classes. Middleware functions and classes are executed in the order of entry in the `settings.py` file.

Study the following pages:

<https://docs.djangoproject.com/en/1.8/topics/http/middleware/>  
<https://docs.djangoproject.com/en/1.8/ref/middleware/>  
<https://docs.djangoproject.com/en/4.2/topics/http/middleware/>  
<https://docs.djangoproject.com/en/4.2/ref/middleware/>  
<https://medium.com/scalereal/everything-you-need-to-know-about-middleware-in-django-2a3bd3853cd6>  
<https://www.tutorialspoint.com/what-is-middleware-in-django>  
<https://raturi.in/blog/understand-and-create-custom-django-middleware/>  
<https://devdocs.io/django~4.1/topics/http/middleware>  
<https://devdocs.io/django~4.1-django-middleware/>

### 7.1.2 Example: Case-insensitive URLs

The regex approach has been dropped from Django 2.1 onward. Instead, we can create a middleware function to achieve the same:

#### 7.1.2.1 middleware.py

In the app folder, create a new file, `middleware.py`, with the following content:

```
from django.http import HttpResponseRedirect

class LowercaseURL:
 def __init__(self, get_response):
 self.get_response = get_response

 def __call__(self, request):
 # Convert the path of the request to lowercase
 request.path = request.path.lower()

 # Redirect to the lowercase URL if necessary
 if request.path != request.path_info:
 return HttpResponseRedirect(request.path)

 response = self.get_response(request)
 return response
```

#### 7.1.2.2 settings.py

Edit the `settings.py` file to include a reference to your custom middleware function:

```
MIDDLEWARE = [
 ...
 "app_replex.middleware.LowercaseURL", #app.file.function
]
```

### 7.1.3 Example: Usage log

In this example, we will log all user actions to a table in the database.

#### 7.1.3.1 Add the model

Add this to `models.py`:

```
class Log(models.Model):
 date_time = models.DateTimeField(blank=True, null=True)
 user = models.ForeignKey('AuthUser', models.DO_NOTHING, blank=True, null=True)
 path = models.TextField(blank=True, null=True)

 class Meta:
 managed = True
 db_table = 'Log'
```

#### 7.1.3.2 Update the database

Migrate

#### 7.1.3.3 middleware.py

In the `app` folder, create a new file, `middleware.py`, with the following content:

```
from datetime import datetime
from django.db import connection

def write_log(get_response):
 # One-time configuration and initialization.

 def middleware(request):
 # Code to be executed for each request before the view is called.

 #Relay the call to the view and get the response
 response = get_response(request)

 # Code to be executed for each request/response after the view is called.
 sql = "INSERT INTO log(date_time, user_id, path) VALUES (%s, %s, %s)"
 cursor = connection.cursor()
 cursor.execute(sql, [datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
 request.user.id, request.path_info])

 #Return the response object
 return response

 return middleware
```

#### 7.1.3.4 settings.py

Edit the `settings.py` file to include a reference to your custom middleware function:

```
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",

 #Custom middleware
 "app_clothing.middleware.write_log", #app.file.function
]

```

#### 7.1.3.5 Test the log

Run the app in a browser and perform some actions as different users. Then, inspect the log file in the MySQL console:

```
SELECT * FROM log;
```

#### 7.1.3.6 Show log in the browser

Add an item for admin users to show the log of actions for (i) a specific user and (ii) a specific date-time interval.

## Usage log

User

S4

Date from

2023 / 04 / 30

Date to

2023 / 05 / 02

Submit

Date and time	User name	Path
2023-04-30 20:08:25	S4	/LoginPost/
2023-04-30 20:08:28	S4	/Products/
2023-04-30 20:08:31	S4	/
2023-04-30 20:08:35	S4	/Products/
2023-04-30 20:08:36	S4	/Login/
2023-04-30 20:08:37	S4	/
2023-04-30 20:08:42	S4	/LoginPost/
2023-05-01 07:07:42	S4	/
2023-05-01 07:07:46	S4	/Login/

Home

See Appendix D for full details.

## 7.2 Mail

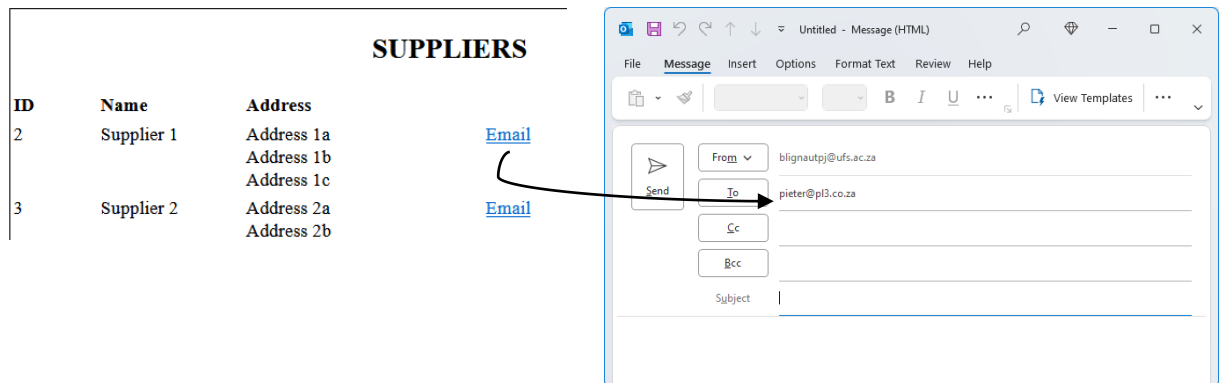
There is a simple way and a better way to send emails through a web page. First, you can use the html mailto tag. This is easy and straightforward but susceptible of spam attacks. Secondly, you can add an HTML form to your site that sends mails through a custom view.

<https://docs.djangoproject.com/en/4.1/topics/email/>  
<https://www.sitepoint.com/django-send-email/>  
<https://devdocs.io/django~4.1/topics/email>

## 7.2.1 mailto

Change the supplier\_list.html template to open the user's registered email client when they click on the email link.

```
<table>
 {% for supplier in supplier_set %}
 <tr valign="top">
 ...
 <td width="80px">Email</td>
 ...
 </tr>
 {% endfor %}
</table>
```



That's all! Unfortunately, web crawlers will find these links and spam your users with emails. View the html source in the browser. The email addresses are open and public for anybody to see – even though we do not display the actual email message on the page.

Reference

- <https://blog.hubspot.com/website/html-mailto>

## 7.2.2 Use an html form

For this demonstration, we will use the customers page.

CUSTOMERS				
ID	Name	Address		Telephone
1	Customer 1	Address 1a	<a href="#">Email</a>	1234
		Address 1b		
2	Customer 2	Address 2a	<a href="#">Email</a>	2345
		Address 2b		
		Address 2c		

### 7.2.2.1 customer\_list.html

Adapt customer\_list.html as follows.

```
<table>
 {% for customer in customer_set %}
 <tr valign="top">
 ...
 <td width="80px">Email</td>
 ...
 </tr>
 {% endfor %}
</table>
```



```

 {% endfor %}
</table>

```

#### Notes

- The actual email address is nowhere referenced on this page.
- When the user clicks on the email link, the user id is sent through as query parameter. A web crawler can do nothing with this unless it has access to the complete database.

#### 7.2.2.2 email.html

When the user clicks on the email link on the customers page, a form is displayed in which the message details can be entered.

```

{% extends "base.html" %}
{% block content %}

<form action="/email_post/{% user.id %}" method="post">
 {% csrf_token %}
 <h2 width="100%" align="center">Send email</h2>

 <table>
 <tr>
 <td>To</td>
 <td>{{ user.username }}</td>
 </tr>
 <tr>
 <td>Subject</td>
 <td><input name="msg_subject"></td>
 </tr>
 <tr>
 <td>Message</td>
 <td><textarea name="msg_content" cols = "40" rows="10"></textarea></td>
 </tr>

 <tr>
 <td colspan="2" align="center">
 <input type="submit" value="Send">
 <input type="button" value="Cancel" onclick="window.location='/'">
 </td>
 </tr>
 </table>
</form>
{% endblock %}

```

#### Notes

- The user object is received from the GET view. This is used to send the user id to the POST view and to display the username in the to field. The actual email address is never displayed.
- If needed, an <input> tag with empty to field can be used which will allow the user to enter any email address.

#### 7.2.2.3 settings.py

In settings.py, add this to the end of the file:

```

EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'mail.pl3.co.za'
EMAIL_HOST_USER = 'pieter@pl3.co.za'
EMAIL_HOST_PASSWORD = '*****'
EMAIL_USE_SSL = False #SSL: False; TLS: True
EMAIL_PORT = 587

```

#### Notes

- The host, user and password depend on your own account.
- If you use SSL instead of TLS, change `EMAIL_USE_SSL = True` and `EMAIL_PORT = 465`.

#### 7.2.2.4 Views

```
from django.conf import settings
from django.shortcuts import render
from django.core.mail import EmailMessage
from . import vw_home, vw_customers
from django.contrib.auth.models import User

def email(request, user_id):
 user = User.objects.get(id = user_id)
 return render(request, 'email.html', {'user': user})

def email_post(request, user_id):
 if request.method == 'POST':
 msg_from = settings.EMAIL_HOST_USER
 user = User.objects.get(id = user_id)
 msg_to = user.email
 msg_subject = request.POST['msg_subject']
 msg_content = request.POST['msg_content']
 recipient_list = [msg_to,]
 msg = EmailMessage(msg_subject, msg_content, msg_from, recipient_list)
 msg.send()
 return vw_customers.customer_list(request)
```

#### Notes

- The GET view receives the user id as parameter, identifies the user and send the entire user object through to the template.
- The POST view also uses the user id in the parameter to determine the email address via the user object. It then retrieves the values that was entered by the user and sends the mail.
- We will show in the following section how to attach files to an email message.

#### 7.2.2.5 Paths

```
from app_clothing.views.vw_email import email, email_post

urlpatterns = [
 path('email/<int:user_id>/', email),
 path('email_post/<int:user_id>/', email_post),
]
```

### 7.2.3 Attach file

#### 7.2.3.1 Template

1. Add the enctype attribute to the form tag. This is essential – don't forget it.

```
<form action="/email_post/{{user.id}}/" method="post" enctype="multipart/form-data">
```

2. Add a `<input type="file">` tag to the form:

```
<tr>
 <td>Attach</td>
 <td><input type="file" name="file" value="Browse ..."></td>
</tr>
```

### 7.2.3.2 View

Adapt the POST view to get the file from the form and attach it.

```
def email_post(request, user_id):
 if request.method == 'POST':
 ...
 msg = EmailMessage(msg_subject, msg_content, msg_from, recipient_list)
 try:
 file = request.FILES['file']
 msg.attach(file.name, file.read())
 except: pass
 msg.send()
 return vw_customers.customer_list(request)
```

### 7.2.3.3 References

- <https://simpleisbetterthancomplex.com/tutorial/2016/08/01/how-to-upload-files-with-django.html>

## 7.3 Files management

Let us develop a facility that a user can use as a warehouse for file – a place where files can be dropped on the server. This means that, if the website is published on the internet, the user can access his files from anywhere.

### 7.3.1 Prepare the server

- Step 1: Create a folder `media` under the `app_` directory.
- Step 2: In `settings.py`, create entries to point to the new media directory. The constants must be named like this.

```
import os
MEDIA_ROOT = os.path.join(BASE_DIR, 'app_clothing/media')
MEDIA_URL = '/media/'
```

Note:

- If you want to use an alternative location for the files, see this: <https://docs.djangoproject.com/en/4.2/topics/files/#file-storage>
- Step 3: In `urls.py`, below the `urlpatterns`, create an entry to map `MEDIA_ROOT` to `MEDIA_URL`:

```
from django.conf import settings
from django.conf.urls.static import static
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```



```

 </td>
 </tr>
</table>

<!-- Uploads -->
<h3 align="left">Upload new file</h3>
<table>
 <tr>
 <td>Previous upload:</td>
 <td>{%if uploaded_file %}
 {{uploaded_file}}
 {%endif%}
 </td>
 </tr>

 <tr>
 <td>Upload file:</td>
 <td><input type="file" name="upload_file" onchange="this.form.submit();"></td>
 </tr>
</table>

<p></p>
<input type="button" value="Home" onclick="window.location='/'">
<p></p>
</form>

{% endblock %}

```

### 7.3.2.2 Delete file

```

{% extends "base.html" %}

{% block content %}

<h2>DELETE FILE</h2>

<form action="/file_delete_post/{{ file_name }}" method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete '{{ file_name }}'?</p>

 <div align="center">
 <input type="submit" class="submit-btn" value=" Yes ">
 <input type="button" class="submit-btn" value=" No "
 onclick="window.location='/files/'">
 <p></p>
 </div>
</form>
{% endblock %}

```

## 7.3.3 Views

Under views, create a file `vw_files`.

### 7.3.3.1 Imports

```

from django.conf import settings
from django.core.files.storage import FileSystemStorage
from django.http import FileResponse
from django.shortcuts import render

import os
from os import listdir
from os.path import isfile, join

```

### 7.3.3.2 List files

```

def uploaded_files():
 media_path = settings.MEDIA_ROOT
 list = listdir(media_path)
 files = []
 for e in list:

```

```

 if isfile(join(media_path, e)):
 files.append(e);
 return files

def files(request):
 return render(request, 'files.html', {'uploaded_files': uploaded_files()})

```

### 7.3.3.3 Upload

```

def files_post(request):
 if request.method == 'POST':
 try:
 file=request.FILES['upload_file']
 f_path = os.path.join(settings.MEDIA_ROOT, file.name)
 if os.path.isfile(f_path): os.remove(f_path)
 fs = FileSystemStorage()
 fs.save(file.name, file)
 return render(request, 'files.html', {'uploaded_file': file.name, \
 'uploaded_files': uploaded_files()})

 except: pass

 return files(request)

```

### 7.3.3.4 Open

```

def file_open(request, file_name):
 f_path = os.path.join(settings.MEDIA_ROOT, file_name)
 if os.path.isfile(f_path):
 content = open(f_path, 'rb')
 response = FileResponse(content)
 return response
 return files(request)

```

### 7.3.3.5 Delete

```

def file_delete(request, file_name):
 return render(request, "file_delete.html", {'file_name': file_name})

def file_delete_post(request, file_name):
 if request.method=="POST":
 f_path = os.path.join(settings.MEDIA_ROOT, file_name)
 if os.path.isfile(f_path):
 os.remove(f_path)
 return files(request)

```

## 7.3.4 Paths

```

from app_clothing.views.vw_files import files, files_post, file_open, \
 file_delete, file_delete_post

urlpatterns = [
 ...

 #Files
 path('files/', files),
 path('files_post/', files_post),
 path('file_open/<str:file_name>/', file_open),
 path('file_delete/<str:file_name>/', file_delete),
 path('file_delete_post/<str:file_name>/', file_delete_post),
]

```

### 7.3.5 References

- <https://simpleisbetterthancomplex.com/tutorial/2016/08/01/how-to-upload-files-with-django.html>

## 7.4 Files in model

A model may have specific types of fields for files and images. These field types contain the file name, content and all attributes. Only the file name with relative path below the media directory is saved in the database. The files themselves are saved under media on the server.

#### 7.4.1 Steps to upload

- Step 1: In a database table, add a field, say `file_name`, where the path to the file will be saved. The type is `VARCHAR` or `TEXT`. This field will save the path and file name where the file will be saved on the server. In the example, add to table `Products` (`description_file_name`).

- Step 2: In the relevant model, add a corresponding file field, for example:

```
description_file = models.FileField(db_column='description_file_name',\
 upload_to='files/')
```

Notes:

- The `FileField` contains the entire file with content and attributes. The database stores only the file name and path.
- The `FileField` field type contains an `upload_to` parameter.
- The `upload_to` directory is a sub-directory of the media directory as specified above. This is where the file will be saved on the server after upload. You don't have to create the directory – Django will do it if it does not exist.
- The file name of a `FileField` includes the subdirectories below `settings.MEDIA_ROOT` – in this case `"/files"`. If we want to retrieve the file name only in the template, add this method to the model:

```
def description_file_name_only (self):
 return os.path.basename(self.description_file.name)
```

- Step 3: In the template, do two things:

1. Add the attribute for enctype to the form:

```
<form action="/product_edit_post/{{ product.product_id }}/"
 method="post"
 enctype="multipart/form-data">
```

2. Add a field to allow the user to upload a file:

```
<tr>
 <td>Description file</td>
 <td><input type="file" name="description_file"></td>
</tr>
```

- Step 4: In the view for POST, retrieve the user entry for the file to be uploaded:

```
import os
from django.conf import settings
def product_edit_post(request, product_id):
 if request.method == 'POST':
 product = Product.objects.get(product_id=product_id)
 ...
 try:
 file= request.FILES['description_file']
 if product.description_file.name:
 f_path = os.path.join(settings.MEDIA_ROOT, product.description_file.name)
 if os.path.isfile(f_path): os.remove(f_path)
```

```

 product.description_file = file
 except: pass

 product.save()
 return product_list(request)

```

#### Notes

- The `'description_file'` parameter above refers to the name of the `<input>` tag as specified in Step 6.
- If another file is uploaded for the same product, the old file is not removed from the uploaded files. You should remove the previous file explicitly.

- Step 5: Display the file content from a template:

```

{% if product.description_file %}
 <a href="{ product.description_file.url }"
 target="_blank">{{ product.description_file_name_only }}
{% endif %}

```

#### Notes

- This will show a clickable link. The response will depend on the browser and the file type. In Firefox, docx will be downloaded and pdf will be displayed in the browser.
- A `FileField` in a model has a `url` property which returns the full path where file is saved.
- If the file has not been defined/assigned, we will get an error. Check for existence.
- The file name of a `FileField` includes the subdirectories below `settings.MEDIA_ROOT` — in this case `"/files"`. So, we used the extra method that we added in the model to return the file name only for display.

- Step 6: Test

- Reload your website and upload a file.
- Inspect the contents of the `media/files` directory.



## 7.4.2 Image files

ImageField is a specialised version of FileField. The same steps apply as for FileField above. In addition / alternatively:

- Step 0: ImageField uses Pillow to ensure that a file is an image.

```
..\env_clothing> pip install pillow
```

- Step 1: In a database table, add a field, say `image_file_name`, where the path to the file will be saved. The type is TEXT.

- Step 2: In the relevant model, add a corresponding image field, for example:

```
image_file = models.ImageField(db_column='image_file_name', \
 upload_to='images/')

def image_file_name_only(self):
 return os.path.basename(self.image_file.name)
```

- Step 3: In the template, do two things:

1. Add the attribute for enctype to the form (if it is not there already):

```
<form action="/product_edit_post/{{ product.product_id }}"
 method="post"
 enctype="multipart/form-data">
```

2. Add a field to allow the user to upload a file:

```
<tr>
 <td>Description file</td>
 <td><input type="file" name="image_file"></td>
</tr>
```

- Step 4: POST

```
import os
from django.conf import settings

try:
 file = request.FILES['image_file']
 if product.image_file.name:
 f_path = os.path.join(settings.MEDIA_ROOT, product.image_file.name)
 if os.path.isfile(f_path): os.remove(f_path)
 product.image_file = file
except: pass
```

- Step 5: Display the image in a template:

```
{% if product.image_file %}
 <td>{{product.image_file_name_only }}</td>
 <td></td>
{% endif %}
```

## 7.4.3 References

<https://docs.djangoproject.com/en/4.2/topics/http/file-uploads/>  
<https://djangocentral.com/uploading-images-with-django/>  
<https://www.geeksforgeeks.org/python-uploading-images-in-django/>  
<https://codinggear.blog/how-to-upload-images-in-django/>  
<https://www.javatpoint.com/django-image-upload>

<https://docs.djangoproject.com/en/4.2/ref/forms/widgets/>

## 7.5 APIs and serialization

In general, an API (Application Programming Interface) provides an interface to set up a communication channel between different software applications or components. Such a channel can refer to the call of executable stubs (exes or dlls), as well as the transfer of data back and forth between various software entities.

Study the following pages:

[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side\\_web\\_APIs/Introduction](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction)

[https://www.w3schools.com/js/js\\_api\\_intro.asp](https://www.w3schools.com/js/js_api_intro.asp)

<https://www.javatpoint.com/create-rest-api-using-django-rest-framework>

<https://www.django-rest-framework.org/>

<https://blog.logrocket.com/the-essential-guide-for-designing-a-production-ready-developer-friendly-restful-api/>

<https://blog.logrocket.com/django-rest-framework-create-api/>

<https://www.tutorialsteacher.com/webapi/what-is-web-api>

<https://www.sankalpjonna.com/learn-django/representing-foreign-key-values-in-django-serializers>

### 7.5.1 REST

For web-driven applications, we need to convert data into a format that can be transferred over the internet. REST (Representational State Transfer) is a framework that provides standards for doing that. It mandates that resources on the web are represented in JSON, HTML, or XML format. So, if we need to transfer the data from a Django app to another app, we need to convert a Django object into one of these formats.

<https://blog.logrocket.com/the-essential-guide-for-designing-a-production-ready-developer-friendly-restful-api/>

<https://www.javatpoint.com/create-rest-api-using-django-rest-framework>

[https://devdocs.io/django\\_rest\\_framework/](https://devdocs.io/django_rest_framework/)

<https://www.codecademy.com/article/what-is-rest>

<https://blog.logrocket.com/django-rest-framework-create-api/>

#### 7.5.1.1 JSON

**JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.  
For example, this is an object with attributes and values.  

```
'{"name":"John", "age":30, "car":null}'.
```
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence. This is a list of three objects:

```
{ "employees": [
 { "firstName":"John", "lastName":"Doe" },
 { "firstName":"Anna", "lastName":"Smith" },
]
```

```
{ "firstName": "Peter", "lastName": "Jones" }
}}
```

It is important to note that attribute names are all represented as strings. Attribute values are either string or numeric.

<https://www.json.org/json-en.html>

#### 7.5.1.2 REST methods

- **GET** is a common method for getting some data from a component. It returns some data from the API based on the endpoint we hit and any parameter we pass.
- **POST** creates new records and updates the newly created record in the database.
- **PUT** takes a new record at the given URI. If the record exists, it updates the record. If record is not available, it creates a new record.
- **PATCH** is used to update one or more data fields.
- **DELETE** deletes the records at the given URI.

<https://www.javatpoint.com/create-rest-api-using-django-rest-framework>

#### 7.5.1.3 HTTP status codes

HTTP status codes are used to let the API consumer know exactly what has happened to process their request. Using it **consistently** is the key to a good API user experience. You don't have to support all the HTTP status codes, but you should try to support the HTTP status codes that align with what your API needs.

Here are some examples of HTTP status codes in the API:

- GET, PUT, PATCH – 200 OK
- POST – 201 Created
- DELETE – 204 No content

The following are a few status codes for errors:

- 400 – Bad request
- 401 – Unauthorized
- 404 – Not found
- 429 – too many requests
- 500 – Internal server error

#### 7.5.1.4 Django REST Framework (DRF)

Django models are Python classes. Data is represented as instances (objects) of these classes. DRF is a REST package on top of Django to translate Django objects into formats like JSON, XML, and vice-versa. This process is known as serialization.

Install REST for your environment. From the terminal:

```
..env_clothing> pip install django_rest_framework
```

In settings.py, add the REST framework under installed apps:

```
INSTALLED_APPS = [
```

```

...
"rest_framework",
"app_clothing.apps.AppClothingConfig",
]

```

<https://www.javatpoint.com/create-rest-api-using-django-rest-framework>  
<https://blog.logrocket.com/django-rest-framework-create-api/>

## 7.5.2 Serialization

Serializers are used to represent the model data in JSON format and convert object instances to a transferable format. On the other hand, de-serializers convert JSON data into a Django object model as an object instance.

Under views, create a file `vw_serializers.py`. Add the following imports and your serializer classes.


```

from rest_framework.views import APIView, status, Response
from rest_framework.response import Response
from rest_framework import serializers

```

### 7.5.2.1 Test data

All our current test data are in tables that contain foreign keys. Serialization for them is a bit tricky, so let us leave that for a bit later and start with a simple example. Add a table, Book, to the database and also to `models.py`.

Table name: Book					
	Name	Data type	Primary Key	Foreign Key	Unique
1	book_id	INTEGER			
2	title	TEXT			
3	author	TEXT			

```

class Book(models.Model):
 book_id = models.AutoField(primary_key=True)
 title = models.TextField(blank=True, null=True)
 author = models.TextField(blank=True, null=True)

 class Meta:
 managed = False
 db_table = 'Book'

```

Add some test data in the table:

	book_id	title	author
1	1	Title A	Author A
2	2	Title B	Author B
3	3	Title C	Author C

### 7.5.2.2 Serializers

A serializer for a model is a class that inherits from `rest_framework.serializers.ModelSerializer`.

The Meta class contains a reference to the model class that we want to refer to as well as a list of fields that must be serialized (converted to JSON format).

```

from django.contrib.auth.models import Book

class BookSerializer(serializers.ModelSerializer):
 class Meta:
 model = Book
 fields = ["book_id", "title", "author"]

```

#### Notes

- It is not necessary that we include all fields as in the model, but if we do, they must be spelled exactly as in the model.
- All names are entered as strings to be compatible to JSON.
- The attributes in the Meta class must be named as model and fields respectively.

### 7.5.3 List View

Views return Response objects that can be displayed in a browser. For DRF, the views are included in a class. We can have views for a list of objects and views to display one object at a time.

#### 7.5.3.1 Views

```
class BookList(APIView):

 def get(self, request, *args, **kwargs):
 users = Book.objects.all()
 serializers = UserSerializer(users, many=True)
 return Response({"status": "success", "users": serializers.data}, status=200)

 def post(self, request, *args, **kwargs):
 #Request data from template and build data object
 data = {
 "title" : request.data.get("title"),
 "author" : request.data.get("author"),
 }

 #Serialize and save
 serializer = BookSerializer(data=data)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_201_CREATED)
 return Response(serializer.errors, status =status.HTTP_400_BAD_REQUEST)
```

#### Notes

- You may not change the names or parameters of the view methods, but you may add parameters for get in BookList.

#### 7.5.3.2 Paths

```
from app_clothing.views.vw_serializers import BookList

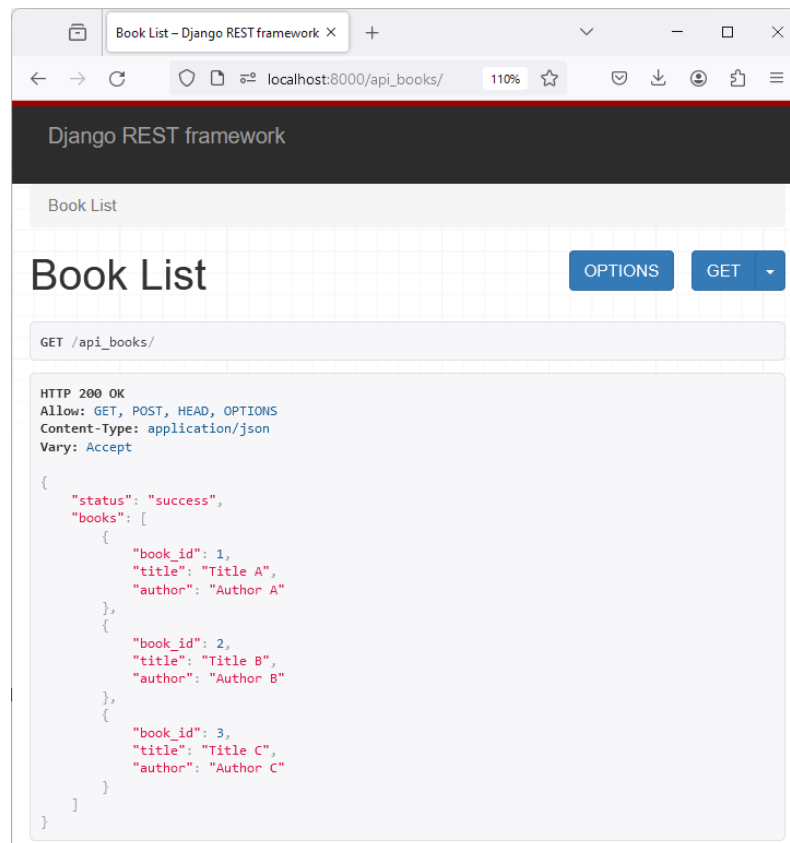
urlpatterns = [
 ...
 path('api_books/', BookList.as_view())
]
```

#### Notes

- BookList is a class, but if we call the method as\_view(), the get method in the class will be called for GET requests.

#### 7.5.3.3 Test GET

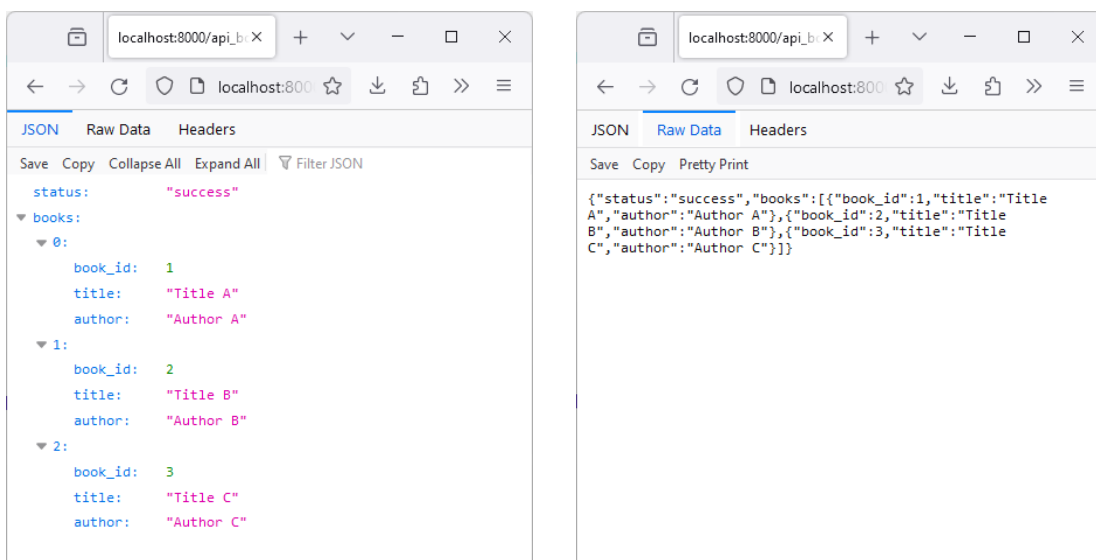
In a browser, type localhost:8000/api\_books and inspect the output.



The GET dropdown allows the user to select a format:



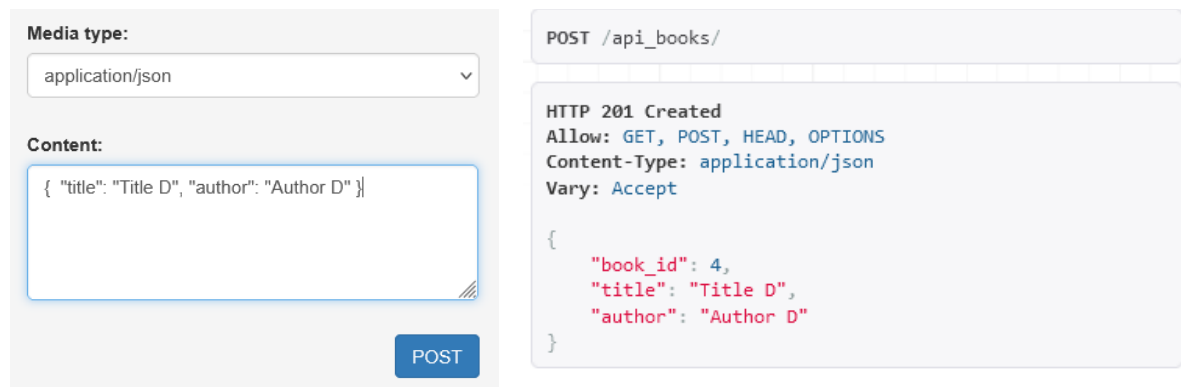
If you choose json, you have options to view the data in a formatted or raw format:



The menu options allow you to save the data to a file or copy to the clipboard. From there the data can be used in another environment.

### 7.5.3.4 Test POST

At the bottom of the screen is a box in which you can enter a new record in JSON format and then add it to the database. Enter data as in the example and click on POST. You should see feedback as on the right. Use SQLite Studio to check the content in the database table.



#### Notes

- We don't assign a specific value for the book\_id. That is an auto-increment field and automatically assigned.

### 7.5.4 Detail View

In `vw_serializers.py`, add a class with methods as indicated below:

```
class BookDetails(APIView):
```

#### 7.5.4.1 GET

```
def get(self, request, book_id, *args, **kwargs):
 try:
 book = Book.objects.get(book_id=book_id)
 serializer= BookSerializer(book) #Note the difference here
 return Response({"status": "success", "book": serializer.data}, status=200)
 except:
 return Response({"res": "Book with the given id does not exist"}, \
 status=status.HTTP_400_BAD_REQUEST)
```

#### 7.5.4.2 PUT

Note the difference between POST and PUT. See the REST methods above again.

```
def put(self, request, book_id, *args, **kwargs):
 try:
 book = Book.objects.get(book_id=book_id)

 data = {
 "title": request.data.get("title"),
 "author" : request.data.get("author")
 }

 serializer = BookSerializer(instance=book, data=data, partial=True)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_200_OK)
 return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
 except:
 return Response({"res": "Book with given id does not exist"}, \
 status=status.HTTP_400_BAD_REQUEST)
```





### 7.5.4.3 DELETE

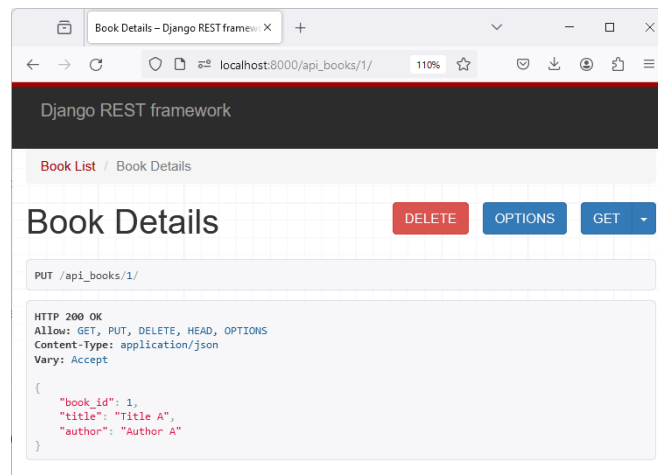
```
def delete(self, request, book_id, *args, **kwargs):
 try:
 book = Book.objects.get(book_id=book_id)
 book.delete()
 return Response({"res": "Object deleted!"}, status=200)
 except:
 return Response({"res": "Book with given id does not exist"}, \
 status=status.HTTP_400_BAD_REQUEST)
```

### 7.5.4.4 Paths

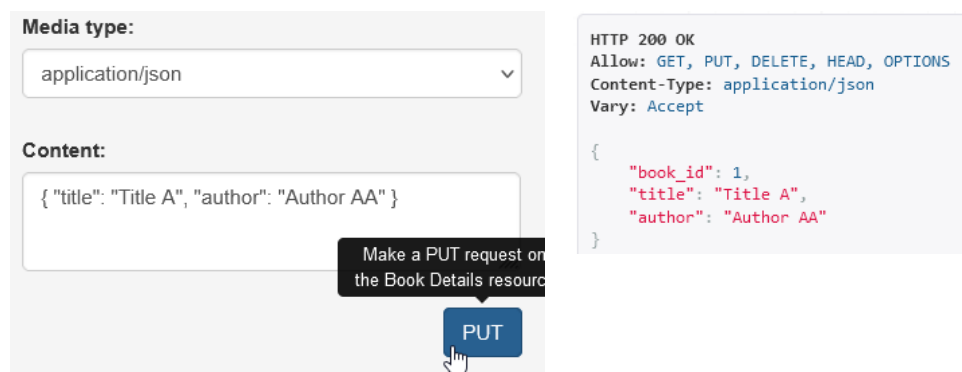
```
urlpatterns = [
 ...
 path('api_books/<int:book_id>', BookList.as_view())
]
```

### 7.5.4.5 Test GET

Specify a book id in the url:



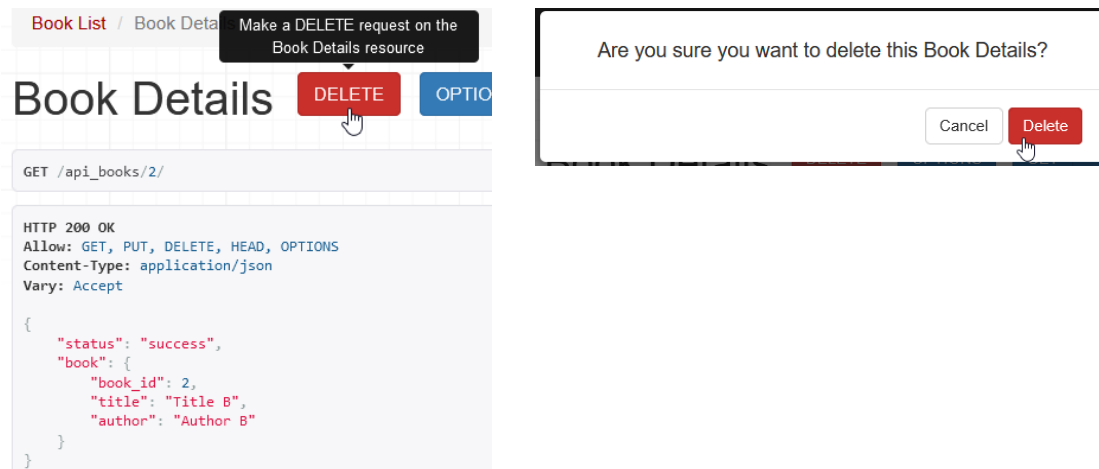
### 7.5.4.6 Test PUT



```
{ "title": "Title BB", "author": "Author BB"}
```

### 7.5.4.7 Test DELETE

A Delete button is added to the browser page. When pressed, a confirmation appears. Check that the book was deleted in the database table.



### 7.5.5 List View with foreign key

There is a one:one relationship between User:Supplier. Remember that in the model, the entire user model is referred to although in the database the foreign key exists of the user\_id only.

#### 7.5.5.1 Serializers

We need to create serializers for both models. We also have two separate serializers for GET and PUT for Supplier.

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

class UserSerializer(serializers.ModelSerializer):
 class Meta:
 model = User
 fields = ["id", "username", "email"]
```

#### 7.5.5.2 GET

```
class SupplierGetSerializer(serializers.ModelSerializer):
 user = UserSerializer()
 class Meta:
 model = Supplier
 fields = ["supplier_id", "user", "supplier_name", "address", "phone_number"]
```

Notes

- Note the reference to the UserSerializer in SupplierGetSerializer.

```
class SupplierList(APIView):

 def get(self, request, *args, **kwargs):
 result = Supplier.objects.all()
 serializers = SupplierGetSerializer(result, many=True)
 return Response({"status": "success", "suppliers": serializers.data}, status=200)
```

Note how the listed fields of user is displayed in th JSON format:



### 7.5.5.3 PUT

```
class SupplierPutSerializer(serializers.ModelSerializer):
 class Meta:
 model = Supplier
 fields = ["supplier_id", "user", "supplier_name", "address", "phone_number"]

class SupplierDetails(APIView):

 def put(self, request, supplier_id, *args, **kwargs):
 try:
 supplier = Supplier.objects.get(supplier_id=supplier_id)
 data = {
 "user": request.data.get("user"),
 "supplier_name": request.data.get("supplier_name"),
 "address" : request.data.get("address"),
 "phone_number": request.data.get("phone_number")
 }

 serializer = SupplierPutSerializer(instance=supplier, data=data, partial=True)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_200_OK)
 return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
 except:
 return Response({"res": "Supplier with given id does not exist"},
 status=status.HTTP_400_BAD_REQUEST)
```

#### Notes

- Although the entire user object is referenced under data, only a specific user\_id is assigned.
- The fact that we have a one:one relationship here can make things difficult to test. You will get an error if you assign a user id to a supplier if that user is already assigned to another supplier. Be careful.

Test the code with a JSON object like this:

```
{ "user" : 5, "supplier_name": "S11", "address": "Address 9", "phone_number": "99" }
```

## 7.6 Maps

<https://pypi.org/project/django-google-maps/>  
<https://cbi-analytics.nl/python-django-google-maps-api-set-up-api-in-google-cloud-platform/>  
<https://cbi-analytics.nl/django-google-maps-tutorial-3-calculate-distance-and-duration-with-distance-matrix-api/>

<https://cbi-analytics.nl/django-google-maps-tutorial-4-creating-a-google-map-maps-javascript-api/>  
<https://cbi-analytics.nl/django-google-maps-tutorial-5-placing-markers-on-a-google-map/>  
<https://github.com/NickMol/Django--GoogleMapsApi/blob/main/google/templates/google/map.html>  
[https://www.tutorialspoint.com/google\\_maps/google\\_maps\\_markers.htm](https://www.tutorialspoint.com/google_maps/google_maps_markers.htm)  
<https://developers.google.com/maps/documentation/javascript/markers>  
<https://developers.google.com/maps/documentation/javascript/reference/marker>

```
pip install googlemaps
```

In `settings.py`:

```
- BASE_COUNTRY = 'ZA'
- GOOGLE_MAPS_API_KEY = 'AIzaSyAS9hfYNRscnT9WlTqzi4R8QhK8lICuM2o'
```

## 7.7 Summary

In this chapter, we covered some of the more advanced topics of website development with Python and Django. We showed how to send an email from within the website, use middleware to record a usage log and use DRF to develop an API so that users can extract and update data from the database. We also showed how to download and upload files from/to the server.

By no means did we cover everything that there is to know about Django. In fact, we touched only the tip of the iceberg - Django is huge. I trust that we triggered your interest to learn more of Django and apply it in everyday applications.

## Chapter 8: Appendix A (Front-end basics)

### 8.1.1 HTML

<https://www.w3schools.com/html/>  
<https://www.tutorialspoint.com/html/index.htm>  
<https://html.com/>

#### 8.1.1.1 html tags

We assume that you have done some html and css in earlier courses, but for the record:

- `<!DOCTYPE html>` informs the browser that the document is to be interpreted as an HTML document.
- `<html></html>` marks the content of an HTML document
- `<head></head>` marks the content of meta (data about data) and includes `<title>`, `<style>`, `<script>` and others.
- `<style></style>` marks the content of style classes. Inside the `<style>` element you specify how HTML elements should be rendered in a browser.
- `<body></body>` defines the document body as opposed to `<header></header>` or `<script> </script>`
- `<div></div>` marks a section in a document that is styled with CSS.
- `<table></table>` marks a table. Tables are good to organise and align content. Tables contain rows `<tr>`. Rows contain cells `<td>`.
- `<hr></h3>` marks a header
- `<ul></ul>` marks an unordered list
- `<li></li>` marks a list item inside an unordered list
- `<a></a>` defines a hyperlink to another page.
- `<input>` allows the user to enter text
- `<input type="password">` allows the user to enter masked text
- `<input type="button" value="OK">` shows a button with the text 'OK'
- `<input type="submit" value="OK">` shows a button with the text 'OK' and submits the form when clicked
- html tags have attributes that determine their appearance, e.g. the width attribute of a table cell.

#### Notes

- NB: Indenting tags is very very important to ensure readability of an html page. It indicates which elements are inside which.
- Your most important shortcuts in VS code will be
  - Ctrl-A (select all)
  - Ctrl-K, Ctrl-F (format selection).
  - F5 (debug) (Or Ctrl-K, W)
- If you want to get details about a specific tag, Google html `<tag name>`, and select the W3Schools link.

### 8.1.1.2 Centre html content

You can use this layout to put your html content in the centre of a browser window.

```
<!DOCTYPE html>
<html>

<head>
 <style>
 .vertical-center { height: 97vh; width: 100vw; display: table-cell; vertical-align: middle; }
 .body-center { text-align: center; }
 .horizontal-center { display: inline-block; border: 1px solid black;
 padding-left: 10px; padding-right: 10px; padding-bottom: 10px;
 margin-left: auto; margin-right: auto }

 </style>
</head>

<body class="body-center">

 <div class="vertical-center">
 <div class="horizontal-center">
 <div align="left">
 Put your content here
 </div>
 </div>
 </div>

</body>

</html>
```

Hint: Use template inheritance to avoid doing this separately for every page.

<https://docs.djangoproject.com/en/4.0/ref/templates/language/#template-inheritance>

### 8.1.2 CSS

<https://www.w3schools.com/css/>  
<https://www.tutorialspoint.com/css/index.htm>  
<https://blog.hubspot.com/website/css-tutorial>

Fonts: [https://developers.google.com/fonts/docs/getting\\_started](https://developers.google.com/fonts/docs/getting_started)

### 8.1.3 JavaScript

<https://www.w3schools.com/js/>  
<https://javascript.info/>  
<https://www.javascripttutorial.net/>  
<https://www.tutorialspoint.com/javascript/index.htm>

## Chapter 9: Appendix B (Python)

### 9.1 Introduction

Python is the language for the server-side development of a Django website. So, before we can proceed, we need to ensure that you are up to speed with Python.

### 9.2 Resources

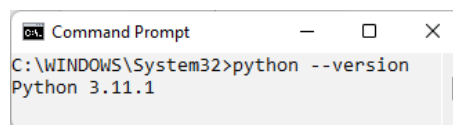
See **eThuto** for several pdf files.

Other resources (in alphabetical order):

- <https://code.visualstudio.com/docs/python/python-tutorial> (For VS Code)
- <https://docs.python.org/3.11/tutorial/> (The official Python document)
- <https://learncodingfast.com/python/>
- <http://www.davekuhlman.org/>
- <http://www.davekuhlman.org/pages/daves-other-stuff.html>
- <http://www.davekuhlman.org/creating-using-python-virtual-environments.html>
- [https://www.digitalocean.com/community/tutorial\\_series/how-to-code-in-python-3](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-python-3)
- <https://www.simplilearn.com/tutorials/python-tutorial>
- <https://www.w3schools.com/python/>

### 9.3 Installation

Check if it is installed on your PC:



```
Command Prompt
C:\WINDOWS\System32>python --version
Python 3.11.1
```

If not, download (<https://www.python.org/downloads/release/python-3111/>) and install.

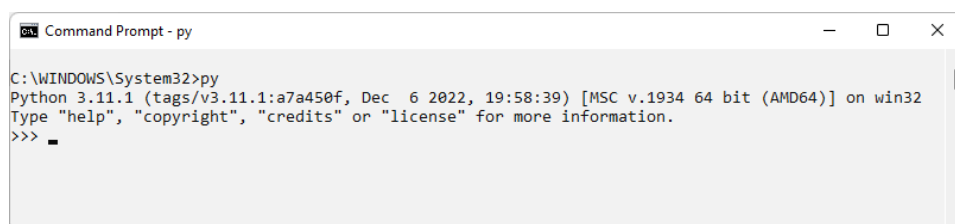
### 9.4 PIP

[https://www.w3schools.com/python/python\\_pip.asp](https://www.w3schools.com/python/python_pip.asp)

### 9.5 Development environments

There are several tools available to write a Python program. You can work directly in a shell from the command prompt, or you can install an editor such as a shell inside Windows, IDLE, PyCharm, or VS Code. We will use VS Code in this course.

#### 9.5.1 Shell



```
Command Prompt - py
C:\WINDOWS\System32>py
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```



## 9.5.2 VS Code

<https://code.visualstudio.com/docs/python/python-tutorial>

### 9.5.2.1 Editor


Zoom: Ctrl+=, Ctrl+-

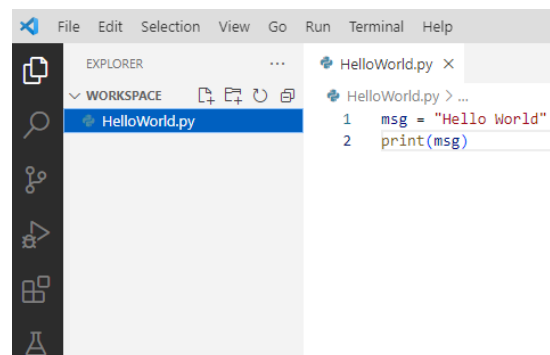
### 9.5.2.2 Extensions

- Django (By Baptiste Darthenay)
- Python (By Microsoft)
- Pylance (Automatically added with Python)

### 9.5.2.3 Create new program

Create new program and run

- Make sure that the Python extension is installed. It should not be necessary to install explicitly as it is available globally.
- File/Open Folder to open a workspace.
  - Create a new folder somewhere on your hard drive where you want your Python files to be saved, e.g. Workspace.
- Select interpreter
  - Ctrl-Shift-P, type Python: Select interpreter, select an installed version
- Create source code file e.g. HelloWorld.py
- Press the button  to run the code in the Terminal window.



### 9.5.2.4 Debugging

- Set breakpoint on a line with F9.
- Press F5
- Debugger configuration: Select Python file ...

### 9.5.2.5 Install and use packages

[https://code.visualstudio.com/docs/python/python-tutorial#\\_install-and-use-packages](https://code.visualstudio.com/docs/python/python-tutorial#_install-and-use-packages)

Packages are libraries.

PyPI is a package index (<https://pypi.org/>)

1. Create virtual environment
  - Open terminal (Ctrl-Shift-`)
  - `py -3 -m venv .venv`
  - Confirm
2. Select environment
  - Ctrl-Shift-P
  - Python: Select Interpreter
  - Select the one with (.venv)
3. Install packages

- `python -m pip install matplotlib`
4. Run program

### 9.5.2.6 Settings

File / Preferences / Settings (Ctrl+,)

### 9.5.2.7 Language mode

Check the language mode in the lower right of the status bar. You might have to toggle between HTML and Django-HTML.

Also see this: <https://stackoverflow.com/questions/61744003/how-can-i-autocomplete-both-html-and-django-html-simultaneously-in-visual-studio/61789506#61789506>

## 9.6 Python basics

This is not an introductory programming course. We assume that you are well versed and have a solid programming background, although it is not necessarily in Python. To bring you up to speed with the Python environment, you can work through one or more of the tutorials listed above under "Resources". I think the w3schools tutorial is probably the best one to start with.

### 9.6.1 Revision topics

Specific topics that you should revise in your own time (self-study) before proceeding with this course, are the following:

- Python syntax. How commands are terminated. The role of indentation.
- Comments
- Variables
- Data types
- Operators
- Lists, Sets, Dictionaries
- Control structures: if, while, for
- Functions
- Arrays
- Classes and objects
  - `__init__()`, `__str__`
- Modules
- RegEx
- PIP
- try ... except
- String formatting

### 9.6.2 Case sensitive

Python is case sensitive and therefore by implication Django as well.

### 9.6.3 Indenting

Instead of braces, we use indenting to indicate that a piece of code belongs inside another construct.

[https://www.w3schools.com/python/python\\_syntax.asp](https://www.w3schools.com/python/python_syntax.asp)

### 9.6.4 Comments

[https://www.w3schools.com/python/python\\_comments.asp](https://www.w3schools.com/python/python_comments.asp)

## 9.6.5 Variables

[https://www.w3schools.com/python/python\\_variables.asp](https://www.w3schools.com/python/python_variables.asp)  
[https://www.w3schools.com/python/python\\_variables\\_global.asp](https://www.w3schools.com/python/python_variables_global.asp)  
[https://www.w3schools.com/python/python\\_scope.asp](https://www.w3schools.com/python/python_scope.asp)

## 9.6.6 Naming conventions

<https://realpython.com/python-pep8/#naming-conventions>  
[https://www.w3schools.com/python/python\\_variables\\_names.asp](https://www.w3schools.com/python/python_variables_names.asp)  
<https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/>

Naming Styles		
The table below outlines some of the common naming styles in Python code and when you should use them:		
Type	Naming Convention	Examples
Function	Use a lowercase word or words. Separate words by underscores to improve readability.	function, my_function
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	x, var, my_variable
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called <i>camel case</i> or <i>pascal case</i> .	Model, MyClass
Method	Use a lowercase word or words. Separate words with underscores to improve readability.	class_method, method

## 9.6.7 Long strings over multiple lines

```
sql = """SELECT D.department_id, department_code, department_name, staff_id,
 CONCAT(title, ' ', surname) AS HOD
FROM Department D LEFT JOIN Staff S ON D.hod_id = S.staff_id
ORDER BY department_name"""
```

Note: Three quotes.

Alternatively, you can use the concat operator with the line end character to concatenate several single-line constants.

```
msg = "Staff number:\t" + staff_number + "\nSurname:\t" + surname \
 + "\nFull names:\t" + full_names + "\nFirst name:\t" + first_name \
 + "\nID number:\t" + id_number + "\nTitle:\t" + title \
 + "\nPosition:\t" + position + "\nOffice number:\t" + office_number \
 + "\nEmail address:\t" + email_address + "\nDepartment:\t" + department
```

Note the back slash at the end of all lines but the last one.

## 9.6.8 Output

[https://www.w3schools.com/python/python\\_variables\\_output.asp](https://www.w3schools.com/python/python_variables_output.asp)  
[https://www.w3schools.com/python/python\\_string\\_formatting.asp](https://www.w3schools.com/python/python_string_formatting.asp)



### 9.6.9 Data types

[https://www.w3schools.com/python/python\\_datatypes.asp](https://www.w3schools.com/python/python_datatypes.asp)  
[https://www.w3schools.com/python/python\\_numbers.asp](https://www.w3schools.com/python/python_numbers.asp)  
[https://www.w3schools.com/python/python\\_casting.asp](https://www.w3schools.com/python/python_casting.asp)  
[https://www.w3schools.com/python/python\\_booleans.asp](https://www.w3schools.com/python/python_booleans.asp)  
[https://www.w3schools.com/python/python\\_datetime.asp](https://www.w3schools.com/python/python_datetime.asp)

### 9.6.10 Strings

[https://www.w3schools.com/python/python\\_strings.asp](https://www.w3schools.com/python/python_strings.asp)  
[https://www.w3schools.com/python/python\\_strings\\_slicing.asp](https://www.w3schools.com/python/python_strings_slicing.asp)  
[https://www.w3schools.com/python/python\\_strings\\_modify.asp](https://www.w3schools.com/python/python_strings_modify.asp)  
[https://www.w3schools.com/python/python\\_strings\\_concatenate.asp](https://www.w3schools.com/python/python_strings_concatenate.asp)  
[https://www.w3schools.com/python/python\\_strings\\_format.asp](https://www.w3schools.com/python/python_strings_format.asp)  
[https://www.w3schools.com/python/python\\_strings\\_escape.asp](https://www.w3schools.com/python/python_strings_escape.asp)  
[https://www.w3schools.com/python/python\\_strings\\_methods.asp](https://www.w3schools.com/python/python_strings_methods.asp)  
[https://www.w3schools.com/python/python\\_string\\_formatting.asp](https://www.w3schools.com/python/python_string_formatting.asp)

### 9.6.11 Operators

[https://www.w3schools.com/python/python\\_operators.asp](https://www.w3schools.com/python/python_operators.asp)

### 9.6.12 Lists

[https://www.w3schools.com/python/python\\_lists.asp](https://www.w3schools.com/python/python_lists.asp)  
[https://www.w3schools.com/python/python\\_arrays.asp](https://www.w3schools.com/python/python_arrays.asp)

### 9.6.13 Conditions and if statements

[https://www.w3schools.com/python/python\\_conditions.asp](https://www.w3schools.com/python/python_conditions.asp)

### 9.6.14 Loops

[https://www.w3schools.com/python/python\\_while\\_loops.asp](https://www.w3schools.com/python/python_while_loops.asp)  
[https://www.w3schools.com/python/python\\_for\\_loops.asp](https://www.w3schools.com/python/python_for_loops.asp)

### 9.6.15 Functions and modules

[https://www.w3schools.com/python/python\\_functions.asp](https://www.w3schools.com/python/python_functions.asp)  
[https://www.w3schools.com/python/python\\_math.asp](https://www.w3schools.com/python/python_math.asp)  
[https://www.w3schools.com/python/python\\_modules.asp](https://www.w3schools.com/python/python_modules.asp)

### 9.6.16 Defensive programming

[https://www.w3schools.com/python/python\\_try\\_except.asp](https://www.w3schools.com/python/python_try_except.asp)

### 9.6.17 User input

[https://www.w3schools.com/python/python\\_user\\_input.asp](https://www.w3schools.com/python/python_user_input.asp)

### 9.6.18 Classes and objects

[https://www.w3schools.com/python/python\\_classes.asp](https://www.w3schools.com/python/python_classes.asp)

#### **9.6.19 JSON**

[https://www.w3schools.com/python/python\\_json.asp](https://www.w3schools.com/python/python_json.asp)

### **9.7 Summary**

In this chapter we introduced you to Python as a programming language. You are responsible to ensure that you are familiar with Python in your own time.



## Chapter 10: Appendix C (Databases)

### 10.1 Example scenario

The content in this chapter will be based on the following scenario.

A clothing shop wants to keep record of its products. Every product is supplied by a supplier. Customers buy products from the shop. All sales must be recorded.

### 10.2 DB Basics

#### 10.2.1 Terminology

A **database system** is essentially nothing more than a computerised record-keeping system. Such a system involves four major components: a **database**, **hardware**, **software** and **users**.

A **database** is an organised collection of **entities** (things) and **relationships** that exist between the entities. The database may be used by the **application programs** of a given enterprise. An enterprise might be a single individual with a very small private database or a complete corporation with a very large shared database (or anything in between).

The **hardware** portion of a database system consists of the secondary storage volumes together with the associated I/O devices as well as the processor(s) and associated main memory that are used to support the execution of the database system software.

Between the physical database and the users of the system is a layer of **software**, the **database management system** (DBMS). All requests from users for access to the database are handled by the DBMS

There are mostly three classes of users involved with a database system: **Application programmers** are responsible for writing application programs that use the database. This is the main concern of this course: Teaching you how to write applications in Python that connects with the physical database. **End users** interact with the database system from online workstations or terminals through either the DBMS or the application programs mentioned above. The third class of user is the **database administrator** (DBA). This person is responsible to create the actual database and to implement the technical controls needed to enforce management's data policies. These policies determine what data should be stored in the database and dictates who can perform what operations on what data in what circumstances.

#### 10.2.2 Entities and relationships

**Entities** are any of the things of interest to an enterprise and about which the enterprise collects data, such as **PRODUCT**, **SUPPLIER**, **CUSTOMER**, **SALE**, etc. (Entities are capitalised when written in normal text.)

There are several kinds of database systems: hierarchical, network, relational and object-oriented. We will focus on relational databases. A **relational database** stores the data for each entity in a **table** with rows and columns. The columns in the table are called **fields**. The rows are referred to as **records**. Each table in a relational database must have a **primary key**, i.e. a field or combination of fields that are guaranteed to be unique from one

record to the other. For example, a customer number (key fields are underlined when written in normal text) can be the key field for a **CUSTOMER**.

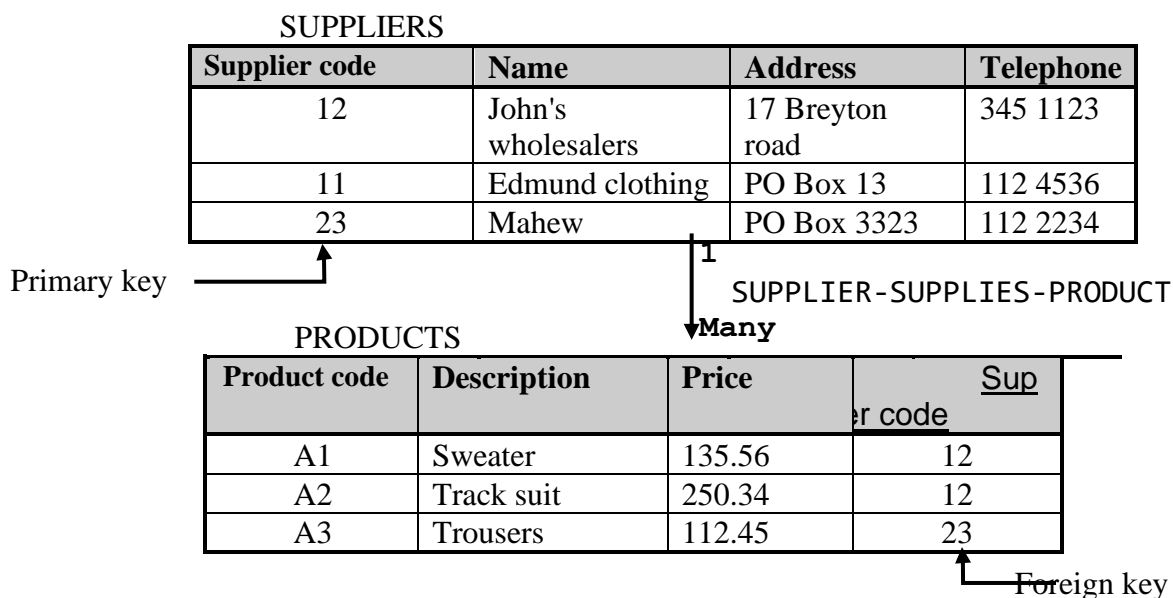
Fields			
<u>Customer number</u>	Name	Address	Telephone
112W3	MPN Jones	PO Box 232, Pietersburg	(015)2115432
2213S	CD Johnson	34 Memory Road, Cape Town	(021)2212321
231A	JP Stopforth	11 Church Street, Bloemfontein	(051)5312345

Records

*CUSTOMERS table with some data*

**Relationships** express real-world associations between entities, e.g. CUSTOMER-BUYS-PRODUCT. This relationship is a **many-to-many relationship** because one customer can buy several products and one kind of product can be bought by more than one customer. We will consider the relationship SUPPLIER-SUPPLIES-PRODUCT to be a **one-to-many relationship** because we assume that a supplier can supply more than one product, but a product can be supplied by one supplier only. The table on the one-side of a one-to-many relationship is called the **primary table** and the table on the many-side is called the **secondary table**.

One-to-many relationships are implemented in a relational database by means of **foreign keys**. When the key of one table appears in another table as a non-key field, it is called a foreign key in the second table. Foreign keys link the records in two tables. In the example tables below, Supplier code, the primary key of SUPPLIERS, appears as foreign key in PRODUCTS. This way it is possible to determine that sweaters and tracksuits are supplied by John's Wholesalers and that Mahew supplies trousers.



Many-to-many relationships cannot be implemented directly in a relational database. We have to create a separate, connecting, table and replace the many-to-many relationship with two one-to-many relationships. For the CUSTOMER-BUYS-PRODUCT relationship, we have to create the table SALE and two one-to-many relationships, CUSTOMERS-SALE and PRODUCTS-SALE. Note that SALE has a compound key that consists of the keys of the linked tables in the many-to-many relationship (Customer number, Product code). This connecting table may have non-key fields, e.g. an invoice number on which the sale was done.

SALE

<u>Customer number</u>	<u>Product code</u>	<u>Invoice number</u>
112W3	A1	1
112W3	A2	1
112W3	A3	1
2213S	A1	2
2213S	A2	3
231A	A3	4

### 10.3 Development environment

We will work with more than one database in this course:

- The clothing shop example. Details are provided below.
- The semester project. The details are provided in the semester project document.
- Summative and formative assessments. The details will be provided at the time.

#### 10.3.1 Local MySQL server

- Download and install MySQL community edition server and Workbench on your computer.
  - <https://dev.mysql.com/downloads/installer/>
  - Select the web-community option. Download.
  - Skip Login or Sign up. Click on “No thanks, just start my download”
- Follow the wizard to install a MySQL server and the Workbench tool
  - In the downloads folder, run the file mysql-installer-web-community-8.0.32.0.msi
  - Under, select the custom option.
- 1. Choosing a Setup type
  - Select Custom. Next.
- 2. Select Products
  - Under MySQL Servers, select the latest version (at the time of this document it was 8.0.32) and drag to the panel of products to be installed.
  - Under applications, select the latest MySQL Workbench and drag to the panel of products to be installed.
  - ~~Under Connectors, select the latest Python connector~~
  - Next
- 3. Download
  - Click on “Execute”.
  - Wait for the downloads to finish.
  - Next
- 4. Installation
  - Execute
  - Next
- 5. Product Configuration

- Next
  - Select defaults. Next.
  - Use Strong Password Encryption. Next.
  - Enter root password. (You must remember this).
  - You do not have to add another user besides the root. Next.
  - Windows Service: Accept defaults. Next.
  - Server File Permissions: Full access. Next.
  - Apply configuration: Execute. Wait for all the green ticks to appear. Finish.
  - Next.
- 6. Installation Complete
    - Finish

### 10.3.2 Database Management System (DBMS)

You need a DBMS to connect to a database server. The DBMS allows you to create and manage a database and add test data. Of course, you can use DDL in the MySQL console, but having a DBMS is so much more convenient. You can use something like MySQL Workbench (free) or DBeaver (free) or Navicat (paid). They are very similar and any one of them can be used to connect to a database on localhost or in the cloud.

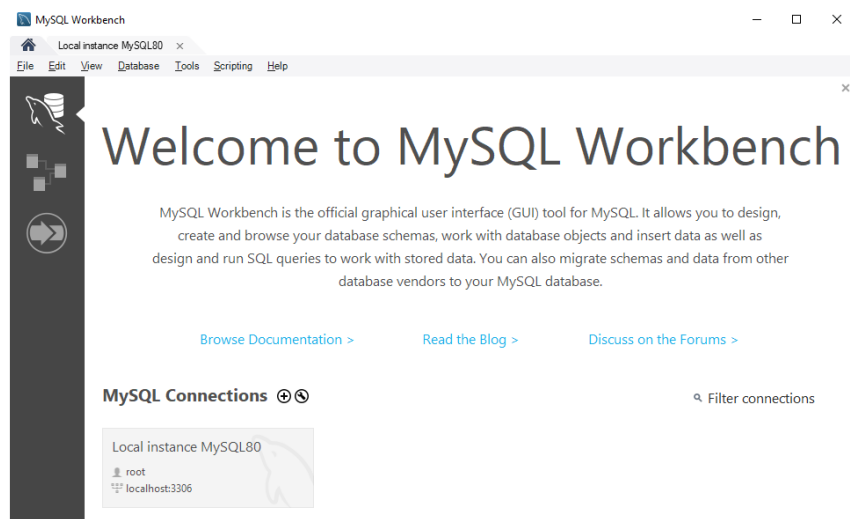
#### 10.3.2.1 MySQL Workbench

Note: You do not have to use MySQL Workbench. You can skip this section and proceed to DBeaver.

Use the MySQL installer to download and install MySQL Workbench. If you followed the procedure in Section 10.3.1 it is done already.

- <https://dev.mysql.com/downloads/installer/>

Run MySQL Workbench from the Start menu. This is what you should see:



- Click on root/ localhost:3306
- Enter password. OK.
- Under schemas, right-click, Create Schema.
  - Name it as sod\_clothing (for the example in this tutorial).
  - Apply, OK, Apply, Finish.
- In the panel on the left, expand the new schema, right-click on Table, then click Create Table.

- Follow the steps to create the tables for the clothing shop example.

#### 10.3.2.2 DBeaver

DBeaver is a free database management system that can be used instead of MySQL Workbench.

Download and install the free community edition.

- <https://dbeaver.io/>
- You may ask for help if you struggle to get this done.

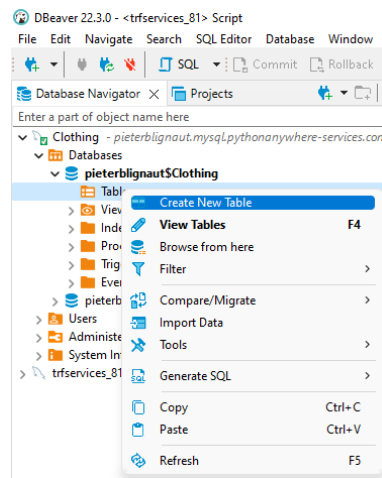
Create the database in DBeaver.

- Right-click on localhost, Create / Database
- Database name: `sod_clothing` (for the example in this tutorial).
- Design the entire database
- Create scripts to create and populate each table.
- Run the scripts on the MySQL console.

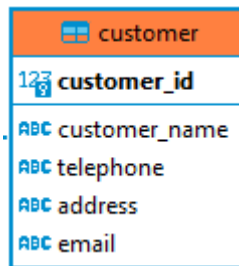
## 10.4 Design and create a database

### 10.4.1 Create the tables

- Right-click on Tables and click "Create New Table".



- Table name: Customer (See the remarks on coding style in the next section.)
- Right click on Columns and click "Create New Column".  
Create columns according to the diagram below.  
The customer\_id column should be marked as primary key.



- Repeat the process for the tables Product, Supplier, and Sale. See the ERD below for details.
- All fields are of type varchar(100) except the ids which are int and product price which must be decimal (10,2).

### 10.4.2 Coding style

We will get to models and views in Django in Chapter 4, but it is important that we design the database according to accepted best Django practices to avoid confusion later.

We will again look at the coding style in Chapter 5, but for now it is important to note that table names should be singular and capitalised (if the server configuration allows it). Field names should all be lower case, using underscores instead of camel case. So, customername or CustomerName is not good. Better is customer\_name.

It does not matter if you have a different opinion - in fact I have regarding the singular form of table names. What matters is that we follow a framework, and we should fall in with the framework to allow others to read and maintain our code.





## 10.4.3 Relationships and referential integrity

### 10.4.3.1 Referential integrity

A **primary key** is a field that will identify a specific record. Primary keys may not be duplicated. It is possible to use the combination of more than one field as primary key. For example, in the Sale table, there may be more than one record with the same customer number and more than one record with the same product code, but the combination of customer number and product code will be unique.

(Emmm ... what happens if a customer comes back the next day and buys another product of the same type? This design does not allow for that possibility.)

**Referential integrity** is a term used to indicate that no values may exist in the secondary table of a relationship if there is not a corresponding value in the primary table. For instance, we may not add a new record to Sale if the corresponding product is not yet registered in Products.

A **foreign key** is a field that can be used to identify a record in a primary table in a secondary table. For example, it is possible to determine the name of supplier of a specific product since we have the supplier code in the Products table. Foreign keys denote (mostly) a one-to-many relationship: In our design, one supplier can supply more than one product, but a specific product comes from one supplier only.

When defining relationships, we need to indicate what should happen with records in a secondary table if the parent record in the primary table is updated or removed.

**On Update: Cascade.** This means that if we change the primary key in the primary table, the foreign keys in all related records in the secondary table must also be changed. For example, if the code of a supplier in the Suppliers table changes, the supplier codes of all related products should be updated - else, we might end up with products for which the supplier is unknown.

**On Delete: Restrict.** This will mean that we cannot delete a record in the primary table if there are still related records in a secondary table. For example, we should not be able to delete a supplier if there are still products for that supplier in the database. If we mark Cascade, all products will be deleted if a supplier is removed which might mean disaster for a business with old stock on hand if a supplier goes out of business.

### 10.4.3.2 Create the foreign keys

Create the foreign keys for Products and Sale.

- For Product:
  - Select Supplier as reference table and supplier\_id as reference column.
- On Delete: Restrict
- On Update: Cascade

	Name	Column	Owner	Ref Table	Type	Ref Object	On Delete	On Update
Columns	product_FK	—	product	supplier	FOREIGN KEY	PRIMARY	Restrict	Cascade
Constraints	123 supplier_id	123supplier_id	—	—	—	supplier_id	—	—
Foreign Keys								
References								

- The auto-generated SQL statement for this operation looks like this. Thank heavens for a good DBMS. Imagine the possible errors if you had to type this in the MySQL console window.

```
ALTER TABLE sod_clothing.product DROP FOREIGN KEY product_FK;
ALTER TABLE sod_clothing.product ADD CONSTRAINT product_FK FOREIGN KEY (supplier_id) REFERENCES
sod_clothing.supplier(supplier_id) ON DELETE RESTRICT ON UPDATE CASCADE;
```

The full DDL for the table looks like this:

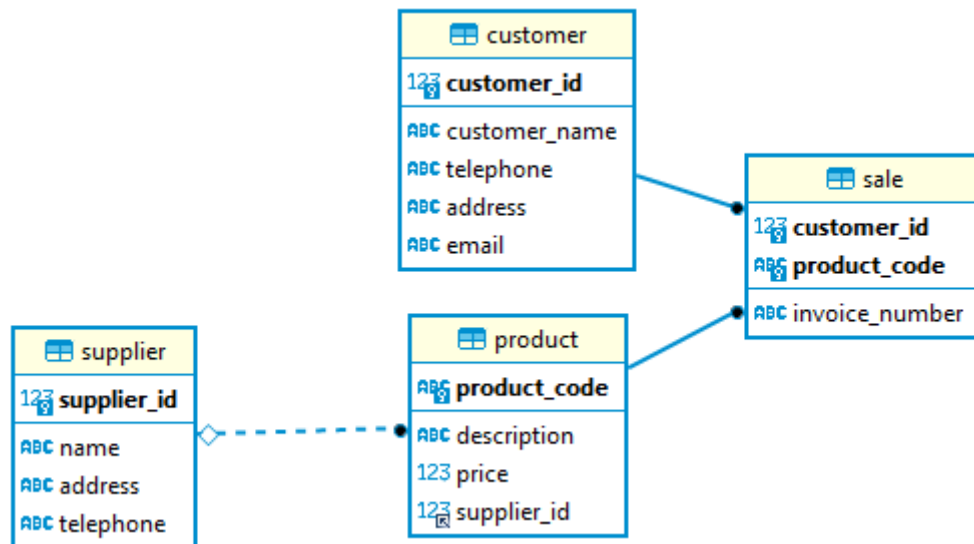
```
CREATE TABLE `product` (
 `product_code` varchar(100) CHARACTER SET utf8mb3 COLLATE utf8mb3_general_ci NOT NULL,
 `description` varchar(100) CHARACTER SET utf8mb3 COLLATE utf8mb3_general_ci DEFAULT NULL,
 `price` decimal(10,2) DEFAULT NULL,
 `supplier_id` int DEFAULT NULL,
 PRIMARY KEY (`product_code`),
 KEY `fk_Supplier_Product` (`supplier_id`) USING BTREE,
 CONSTRAINT `product_FK` FOREIGN KEY (`supplier_id`) REFERENCES `supplier` (`supplier_id`) ON
DELETE RESTRICT ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

#### 10.4.3.3 Create the foreign keys

- Sale is a connection table which facilitates a more:more relationship between customer and product. That means that one customer can buy more than one product and one type of product can be bought by more than one customer. For every combination of customer and product (sale), we have an invoice.
- Since we cannot have combined keys in Django, it is good practice to add an auto increment key field. In Django, we can specify the `unique_together` attribute for the field combination that we want to be unique, in this case `customer_id` and `product_code`.
- Create two foreign keys on the Sale table - one to reference the customer and another to reference the product.

#### 10.4.3.4 Create the ERD

- Double click on the database name. A tab, ER Diagram, will open and the existing tables will be added automatically. Connect the tables as in the diagram below.
- If the foreign keys were added correctly, the links should be drawn automatically.
- Hints:
  - It is good practice to set up an ERD such that the one-to-many relationships are read from left to right. Also, relationships should not cross over. This allows for easier readability and maintenance.



Note:

- To create an ERD in MySQL Workbench, click on Database / Reverse Engineer... and follow the steps.

#### 10.4.4 Enter test data

- Enter some test data in all the tables and check that all works fine.
- You can enter the data either through DBeaver or with an SQL INSERT statement in the MySQL console, for example:

```
INSERT INTO Supplier (supplier_id, name, address, telephone)
VALUES (1, 'Supplier 1', '12 Church street', '0123456789');
```

- Hint:
  - For easy reference, enter data that can be easily recognised, for example, Customer 1, Customer 2, Supplier 1, Supplier 2, etc.
- Try to add a product with a supplier that does not exist and check that the restriction works.
- Change the code of a product and check that the change is cascaded to Sale.
- Try to delete a record in the Suppliers table and check that the restriction works.

#### 10.5 SELECT Queries

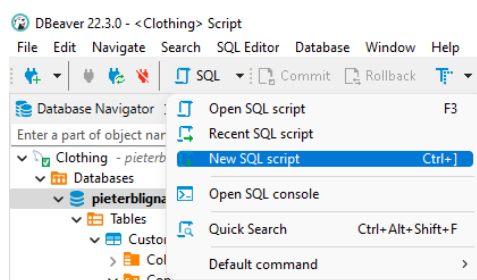
Suppose the shop manager needs the following information from the database:

- Q1: For the purposes of a catalogue, the manager needs an alphabetical list of all the products with their codes, descriptions and prices. Only products with a price of R200 or less must be listed.
- Q2: The manager needs a list of all product details, including the respective supplier names and telephone numbers.
- Q3: The manager receives a query from MPN Jones and needs a list of all the product descriptions and the respective supplier names for all the products bought by him.
- Q4: What is the total amount on invoice 1?

Given tables with a relatively small amount of data, a human can answer the above queries by searching the tables manually. This is, however, not always the case and we need a way to query the database programmatically. For this purpose, a query language, called **SQL (Structured Query Language)** (some pronounce it "es kew el", others pronounce it as "sequel") was developed. **SQL** is a platform independent language that uses the same syntax irrespective of the application.

##### 10.5.1 A single table query

Open the SQL editor in DBeaver or Navicat.



- The first question above can be answered by the following SQL statement:  

```
SELECT product_code, description, price
FROM Product
```

```
WHERE price <= 200
ORDER BY description
```

- Click on the green arrow to run the script.
- An SQL SELECT statement has the following general structure:

```
SELECT <field names> [, <aggregate functions>]
FROM <table names>
[WHERE <conditions>]
[ORDER BY <field names> [DESC]]
[GROUP BY <field names>]
```

- Note the following:
  - Keywords are written in capitals.
  - <> indicates elements that must be listed from the specific database.
  - [ ] indicates optional elements.
- Note that if there are one or more spaces in a field name, the field name must be enclosed in reverse quotes.
- SQL is not case sensitive although we have some conventions:
  - Keywords are written in all capitals
  - Table names are capitalised
  - The various parts (SELECT, FROM, WHERE, ORDER BY, GROUP BY) are written on separate lines.

### 10.5.2 Join tables

- Add a new query and enter the following SQL statement to answer the second question (Q2):

```
SELECT Product.*, name, address, telephone
FROM Supplier S
 INNER JOIN Product P ON S.supplier_id = P.supplier_id
```

- Note the following:
  - A \* may be used in the **SELECT** clause to include all the fields from a table. If the same field name is used in more than one table, the field name must be preceded with the table name and a period.
  - Since the fields that must be included are from two tables, they must be joined in the **FROM** clause. The **ON** clause specifies the fields from the two tables that must be equal. Refer to the ERD above again.
  - If the field names in the two tables are the same, we can use **USING** instead of **ON**.

```
SELECT Products.*, name, address, telephone
FROM Supplier
 INNER JOIN Product USING (supplier_id)
```

- The third query (Q3) is a little bit more involved and may look like this:

```
SELECT description, S.name as `Supplier name`
FROM Customer C INNER JOIN Sale ON C.customer_id = Sale.customer_id
 INNER JOIN Product P ON Sale.product_code = P.product_code
 INNER JOIN Supplier S ON P.supplier_id = S.supplier_id
WHERE customer_name = "MPN Jones";
```

- Note the following:
  - It is always handy to have a printed version of the ERD nearby when you write queries.

- All the table names that are involved in either the SELECT statement or in the conditions must be joined in the FROM statement.
- To resolve the ambiguity between customer name and supplier name for a user, an alias is defined. Since the alias has a space, it is enclosed in reverse quotes.
- Aliases can be defined for table names to make the writing in subsequent references to the table somewhat less.

### 10.5.3 Aggregate functions

- Add a new query and enter the following SQL statement to answer the fourth question.

```
SELECT SUM(price) AS TotalPrice
FROM Product P INNER JOIN Sale S USING(product_code)
WHERE invoice_number = "1"
```

- We can expand the query a bit to list the totals on all invoices:

```
SELECT invoice_number, SUM(price) AS TotalPrice
FROM Product P INNER JOIN Sale S USING(product_code)
GROUP BY invoice_number
```

- Note the following:
  - Columns can be named with an appropriate alias after the keyword AS.
  - All fields that are not included in the aggregate function(s) must be included in a GROUP BY clause.
  - Aggregate functions can be used to determine various statistics on data fields. Other functions that are available are AVG, COUNT, MIN, MAX, SUM and STD. GROUP\_CONCAT is also very handy.
- Consult the official MySQL documentation to get help on what is available: <https://dev.mysql.com/doc/refman/8.0/en/>

## 10.6 Python and MySQL

### 10.6.1 Get started

[https://www.w3schools.com/python/python\\_mysql\\_getstarted.asp](https://www.w3schools.com/python/python_mysql_getstarted.asp)

### 10.6.2 Create database

[https://www.w3schools.com/python/python\\_mysql\\_create\\_db.asp](https://www.w3schools.com/python/python_mysql_create_db.asp)

### 10.6.3 Create and drop table

[https://www.w3schools.com/python/python\\_mysql\\_create\\_table.asp](https://www.w3schools.com/python/python_mysql_create_table.asp)  
[https://www.w3schools.com/python/python\\_mysql\\_drop\\_table.asp](https://www.w3schools.com/python/python_mysql_drop_table.asp)

### 10.6.4 Insert data

[https://www.w3schools.com/python/python\\_mysql\\_insert.asp](https://www.w3schools.com/python/python_mysql_insert.asp)

### 10.6.5 Select

[https://www.w3schools.com/python/python\\_mysql\\_select.asp](https://www.w3schools.com/python/python_mysql_select.asp)  
[https://www.w3schools.com/python/python\\_mysql\\_where.asp](https://www.w3schools.com/python/python_mysql_where.asp)

[https://www.w3schools.com/python/python\\_mysql\\_orderby.asp](https://www.w3schools.com/python/python_mysql_orderby.asp)  
[https://www.w3schools.com/python/python\\_mysql\\_limit.asp](https://www.w3schools.com/python/python_mysql_limit.asp)  
[https://www.w3schools.com/python/python\\_mysql\\_join.asp](https://www.w3schools.com/python/python_mysql_join.asp)

#### **10.6.6 Delete**

[https://www.w3schools.com/python/python\\_mysql\\_delete.asp](https://www.w3schools.com/python/python_mysql_delete.asp)

#### **10.6.7 Update**

[https://www.w3schools.com/python/python\\_mysql\\_update.asp](https://www.w3schools.com/python/python_mysql_update.asp)

### **10.7 Semester project, Part 2**

See the separate document on the semester project for details.

### **10.8 Summary**

In this chapter, we revised some basic database concepts and discussed the creation of a database. We also referred to the usage of a DBMS to design a database and manage data. We discussed SQL as a query language to manage data in a database and showed some examples to retrieve data. We also referred to the embedding of SQL statements from within Python code.

## Chapter 11: Appendix D (Representative examples)

Replace repex with the name of your app.

### 11.1 New project

A project can be seen as a site. On your PC, under the folder where you want to create the product:

```
$ py -m venv env_repex
..\env_repex\scripts> activate ← NB!!
..\env_repex> pip install Django==4.1.7
..\env_repex> Django-admin startproject pr_repex
```

### 11.2 New app

A project can have multiple apps, each pointing to a different sub-domain.

```
..\env_repex> cd pr_repex
..\env_repex\pr_repex> py manage.py startapp app_repex
```

NB: Specification of the version of Django is essential as later versions are not compatible with the version of MySQL that is used on PythonAnywhere and Riptide.

See later (sub-domains) for a discussion of how to create multiple apps under the same project.

### 11.3 VS Code

- Open the folder for env\_repex in VS Code
- Select Python interpreter  
(Ctrl-Shift-P/Python: Select interpreter / ./Scripts/python.exe)
- Under the app\_repex folder:
  - Delete the file views.py
  - Create file middleware.py
  - Create folders: templates, views, media, static
  - Under ..\templates, create a folder home
  - Under ..\templates\home, create files, base.html, index.html and message.html.
  - Under ..\views, create a file, vw\_home.py.

### 11.4 settings.py

#### 11.4.1 Minimum changes from default

```
import os

ALLOWED_HOSTS = ['*']

INSTALLED_APPS = [
 ...
 'app_repex.apps.AppRepexConfig'
]

TEMPLATES = [
 {
 'BACKEND': ...,
 'DIRS': [os.path.join(BASE_DIR, "app_repex/templates/home"),
 ...
```



```
],
 },
]
```

Comment out all entries under `AUTH_PASSWORD_VALIDATORS` during debugging.

## 11.4.2 Complete

The below content is an example of a complete settings file.

```
#Paths
from pathlib import Path
import os
BASE_DIR = Path(__file__).resolve().parent.parent
MEDIA_ROOT = os.path.join(BASE_DIR, 'app_replex/media')
MEDIA_URL = '/media/'

WSGI_APPLICATION = 'pr_replex.wsgi.application'
ROOT_URLCONF = 'pr_replex.urls'
STATIC_URL = 'static/'
ALLOWED_HOSTS = ['*']

DEBUG = True #Error messages are shown in the browser. Change to false for production version.

#Data formats
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'

#Other configs
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'app_replex.apps.AppReplexConfig'
]

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',
]

TEMPLATES = [
 {
 'BACKEND': 'django.template.backends.django.DjangoTemplates',
 'DIRS': [os.path.join(BASE_DIR, "app_replex/templates/home"),
 os.path.join(BASE_DIR, "app_replex/templates/item")],
 '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',
],
 },
 },
]

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

#Security
```

```

SECRET_KEY = 'django-insecure--9u-z#4sne!g%-imfb8=apa*9a@rm-%r5ez^d6cq8aslz2ot2='
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',
 # },
]

#For emails
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'mail.pl3.co.za'
EMAIL_USE_SSL = False
EMAIL_PORT = 587
EMAIL_HOST_USER = 'pieter@pl3.co.za'
EMAIL_HOST_PASSWORD = '*****'

```

## 11.5 Run server and test

```

Activate
..\env_replex\pr_replex> py manage.py runserver
http://localhost:8000

```

## 11.6 Database

You can stay with the default SQLite database or opt to create a MySQL database.

### 11.6.1 For MySQL:

- Install MySQL  

```
..\env_replex\pr_replex> pip install mysql
```
- Change settings.py.

```

DATABASES = {
 "default": {
 "ENGINE": "django.db.backends.mysql",
 "NAME": "<database name>", #Case sensitive
 "USER": "<username>",
 "PASSWORD": "<user password>",
 "HOST": "localhost",
 "PORT": "3306",
 "OPTIONS": {
 "init_command": "SET sql_mode='STRICT_TRANS_TABLES'",
 }
 }
}

```

### 11.6.2 For SQLite:

Use SQLite Studio and add database ..\env\_name\pr\_name\db.sqlite3.  
 Make sure to provide a representative name to appear in the list.

### 11.6.3 Migrate

```
..\env_name\pr_name>python manage.py migrate
```

Check that the admin tables were added to the database (There should be 10.)

### 11.6.4 Synchronise

I prefer to not add other tables at this stage. I would rather define the models and then migrate to create the tables automatically. Make sure that `managed=True` for new models that you add.

If you do add the tables now, note the following:

- Add tables as needed, starting with primary tables (tables on the 1-side of relationships).
  - Table names must be lowercase, singular. Use camel case for compound table names, e.g. `usageLog`.  
Field names must be lowercase, separate words with `_`.
  - All keys must be `INT(EGER)`, preferable named `tablename_id`.  
Configure as `autoincrement`, Not `NULL`.
  - String fields must have type `TEXT`.  
Phone numbers, student numbers, ID numbers, etc. are `TEXT`. For ID numbers, name the field as `id_number` to distinguish from `tablename_id`.  
Amounts must have type `DECIMAL(10,2)`  
Dates must have type `DATE`.
  - Make sure all foreign keys are properly defined.
  - Use DBeaver to draw an ERD. Print the ERD and refer to the hard copy during debugging.
- Update models  
`..\env_name\pr_name>py manage.py inspectdb > app_name/models.py`  
Alternatively:
    - `..\env_name\pr_name>py manage.py inspectdb [table name]`
    - Copy the model from the terminal and paste into `models.py`

## 11.7 Basic templates, views and paths

### 11.7.1 base.html

```
<!DOCTYPE html>
<html>

<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">

 <link rel="stylesheet"
 href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.1/css/all.min.css">

 <title>SOD517C</title>

 <style>
 body { text-align: center;}
 .div_center { display:inline-block; border: 1px solid;}
 ul, li, table { text-align: left; }
 </style>

</head>

<body>
 <div class="div_center">
 {% block content %}
 {% endblock content %}
 </div>
</body>
</html>
```

### 11.7.2 index.html

```
{% extends "base.html" %}
{% block content %}
 <h2> Header </h2>

 Items

{% endblock %}
```

### 11.7.3 message.html

```
{% extends "base.html" %}
{% block content %}

 <h2>MESSAGE</h2>
 <pre>{{ message }}</pre>
 <div align="center">
 <input type="button" value="OK" text-align="center" onclick="window.location='/'">
 </div>

{% endblock %}
```

### 11.7.4 settings.py

We need to tell Django where to find the templates. Add an appropriate entry under TEMPLATES everytime that you add a folder under `..\app_\templates`.

```
TEMPLATES = [
 {
 'BACKEND': ...,
 'DIRS': [os.path.join(BASE_DIR, "app_replex/templates/home"),
 ...
],
 },
]
```

### 11.7.5 vw\_home.py

```
from django.shortcuts import render

def index(request):
 return render (request, 'index.html')

def show_message(request, msg):
 return render (request, "message.html", {'message': msg})
```

### 11.7.6 urls.py

```
from django.contrib import admin
from django.urls import path
from app_name.views import vw_home

urlpatterns = [
 path('admin/', admin.site.urls),
 path('', vw_home.index),
 path('index/', vw_home.index),
] #urlpatterns
```

Make sure to import all view files as they are created.

## 11.8 Users

Create superuser:

```
..\pr_replex> python manage.py createsuperuser
```

## 11.9 Models

### 11.9.1 Model for primary table

```
class Person(models.Model):
 person_id = models.AutoField(primary_key=True)
 id_number = models.TextField(blank=True, null=True)
 name_surname = models.TextField()
 address = models.TextField(blank=True, null=True)
 email = models.TextField(blank=True, null=True)

 dob = models.DateField(blank=True, null=True)
 is_adult = models.BooleanField(blank=True, null=True)
 gender = models.TextField(blank=True, null=True)

 cv_file = models.FileField(db_column='cv_file_name', upload_to='files/', blank=True, null=True)
 photo_file = models.ImageField(db_column='photo_file_name', upload_to='images/')

 class Meta:
 managed = True
 db_table = 'Person'

 def cv_file_name_only(self):
 return os.path.basename(self.cv_file.name)

 def photo_file_name_only(self):
 return os.path.basename(self.photo_file.name)

 def age(self):
 return datetime.now.year - dob.year;
```

### 11.9.2 Model for this example

The examples below are based on the following model.

```
class Item(models.Model):
 item_id = models.AutoField(primary_key=True)
 item_name = models.TextField(blank=True, null=True)
 item_date = models.DateField()

 class Meta:
 managed = True
 db_table = 'Item'
```

- Migrate

```
..\env_name\pr_name>python manage.py makemigrations
..\env_name\pr_name>python manage.py migrate
```

### 11.9.3 Model for secondary table (With foreign key)

```
class Vehicle(models.Model):
 vehicle_id = models.AutoField(primary_key=True)
 registration_number = models.TextField(blank=True, null=True)
 registration_file = models.FileField(db_column='registration_file_name', upload_to='files/', \
 blank=True, null=True)
 make = models.TextField(blank=True, null=True)
 model = models.TextField(blank=True, null=True)
 owner = models.ForeignKey('Person', models.DO_NOTHING, blank=True, null=True)
 year = models.IntegerField()
 price = models.DecimalField(max_digits=10, decimal_places=2, blank=True, null=True)

 class Meta:
 managed = True
 db_table = 'Vehicle'

 def registration_file_name_only(self):
 return os.path.basename(self.registration_file.name)
```

Notes:

- If migrate cannot create foreign keys, it is probably because of conflicting data types in the database. Create the relationship in the database and then pipe the tables back to models.

#### 11.9.4 Notes on models and model fields

- For ImageField, install Pillow
- Remember to migrate after new models were added or changes to the models were made.

### 11.10 CRUD templates, views and paths

Disclaimer: You may copy and paste from the material below for the project or during tests, but I take no responsibility for any errors that may be present in the code.

#### 11.10.1 Folders

- Under templates, create a separate folder for every entity. So, for this example, there is a folder `..\app_name\templates\item`.
- Under views, create a file for every entity. So, for this example, there is a file, `..\app_name\ views\vw_item.py`.

#### 11.10.2 item\_list.html

```
{% extends "base.html" %}
{% block content %}

<h1>Items</h1>

<!-- Column headers -->
<table>
 <tr>
 <th width="150px" align="left">Name</th>
 <th width="150px" align="left">Date</th>
 <td>Add item</td>
 </tr>
</table>

<!-- Data rows -->
<table>
 {% for item in items %}
 <tr>
 <td width="150px">{{ item.item_name }}</td>
 <td width="150px">{{ item.item_date|date:'d F Y' }}</td>
 <!-- https://docs.djangoproject.com/en/5.0/ref/templates/builtins/#date -->
 <td>Edit</td>
 <td>Delete</td>
 </tr>
 {% endfor %}
</table>

<!-- Buttons -->
<div align="center">
 <input type="button" value=" Home " onclick="window.location='/'">
</div>

{% endblock %}
```

#### 11.10.3 item\_add.html

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

{% block content %}

<h1>Add item</h1>
<form action="/item_add_post/" method="post">
```

```

{% csrf_token %}
<table>
 <tr>
 <td>Item name</td>
 <td><input type="text" name="name" ></td>
 </tr>

 <tr>
 <td>Date</td>
 <td><input type="date" name="date" ></td>
 </tr>

 <tr><td>
</td></tr>
 <tr>
 <td></td>
 <td>
 <input type="submit" value=" Submit ">
 <input type="button" value=" Cancel " onclick="window.location='/item_list'">
 </td>
 </tr>
</table>
<p></p>
</form>

{% endblock %}

```

#### 11.10.4 item\_edit.html

```

{% extends "base.html" %}

{% block content %}

<h1>Edit item</h1>
<form action="/item_edit_post/{{item.item_id}}/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Item name</td>
 <td><input type="text" name="name" value="{{item.item_name}}" ></td>
 </tr>

 <tr>
 <td>Date</td>
 <td><input type="date" name="date" value="{{item.item_date|date:'Y-m-d'}}" ></td>
 </tr>

 <tr><td>
</td></tr>
 <tr>
 <td></td>
 <td>
 <input type="submit" value=" Submit ">
 <input type="button" value=" Cancel " onclick="window.location='/item_list'">
 </td>
 </tr>
 </table>
 <p></p>
</form>

{% endblock %}

```

#### 11.10.5 item\_delete.html

```

{% extends "base.html" %}

{% block content %}

<h1>Delete item</h1>

<form action="/item_delete_post/{{item.item_id}}/" method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete Item {{ item.name }}?</p>

 <div align="center">
 <input type="submit" value=" Yes ">
 <input type="button" value=" No " onclick="window.location='/item_list/'">
 </div>
</form>

```

```

 <p></p>
 </form>
{% endblock %}

```

## 11.10.6 vw\_item.py

This views file includes very typical and basic views. Your needs may be much more complex.

```

from django.shortcuts import render
from ..models import Item
from ..views.vw_home import index, show_message

def item_list(request):
 items = Item.objects.all()
 return render(request, 'item_list.html', {'items': items})

def item_add(request):
 return render(request, 'item_add.html')

def item_add_post(request):
 if request.method=="POST":
 name = request.POST['name']
 date = request.POST['date']
 item = Item(item_name=name, item_date=date)
 item.save()
 return item_list(request)

def item_edit(request, item_id):
 item = Item.objects.get(item_id=item_id)
 return render (request, 'item_edit.html', {'item': item})

def item_edit_post(request, item_id):
 if request.method=="POST":
 item = Item.objects.get(item_id=item_id)
 item.item_name = request.POST['name']
 item.item_date = request.POST['date']
 item.save()
 return item_list(request)

def item_delete(request, item_id):
 item = Item.objects.get(item_id=item_id)
 return render (request, 'item_delete.html', {'item':item})

def item_delete_post(request, item_id):
 if request.method=="POST":
 item = Item.objects.get(item_id=item_id)
 item.delete()
 return item_list(request)

```



## 11.10.7 urls.py

```
from django.urls import path
from app_bookings.views import vw_home, vw_item

urlpatterns = [
 ...
 #Item
 path('item_list/', vw_item.item_list),
 path('item_add/', vw_item.item_add),
 path('item_add_post/', vw_item.item_add_post),
 path('item_edit/<int:item_id>/', vw_item.item_edit),
 path('item_edit_post/<int:item_id>/', vw_item.item_edit_post),
 path('item_delete/<int:item_id>/', vw_item.item_delete),
 path('item_delete_post/<int:item_id>/', vw_item.item_delete_post),
]
```

## 11.11 Input tags in html

### 11.11.1 Text

```
<input type="text" value="{{person.name}}" name="name" required>
```

In view:

```
person.name_surname = request.POST['name_surname']
```

### 11.11.2 TextArea

```
<textarea name="description" rows="5" cols="60">{{product.description}}</textarea>
```

### 11.11.3 Integer

```
<input type="number" min="0" step="1" name="year" value="{{ vehicle.year }}" size="5px">
```

In view:

```
vehicle.year = int(request.POST['year'] or 0)
```

### 11.11.4 Decimal

```
<input type="number" min="0.00" step="any" name="price" value="{{vehicle.price}}">
```

In view:

```
from decimal import Decimal
vehicle.price = Decimal(request.POST['price'] or 0.00)
```

### 11.11.5 Date / DateTime

```
<input type="date" name="dob" value="{{person.dob|date:'Y-m-d'}}">
<input type="datetime-local" name="datetime"
 value="{{document.signed_on|date:'Y-m-d H:i:s'}}">
```

In view:

```
dt = request.POST['date_taken']
if dt != '': person.dob = request.POST['date_taken']
else: person.dob = None
```

In model: Make sure that the field type is DateTimeField – not TextField.

### 11.11.6 Radio button

```
<input type="radio" name="gender" value="Male"
 {% if person.gender == "Male" %} checked {% endif %}> Male
```

```
<input type="radio" name="gender" value="Female"
 {% if person.gender == "Female" %} checked {% endif %}> Female
```

In view:

```
person.gender = request.POST['gender']
```

### 11.11.7 Checkbox (single)

```
<input type="checkbox" name="is_adult"
 value="True"
 {% if person.is_adult %} checked {% endif %} >
```

In the view:

```
#Dict key will only be present if the box is checked
try: person.is_adult = request.POST['is_adult']
except: person.is_adult = False
```

### 11.11.8 Checkboxes in a list

```
<input type="checkbox" name="is_adult" value="adult"
 {% if person.is_adult %} checked {% endif %} >
```

In a view, check if the value is present in all checked checkboxes with the same name.  

```
person.is_adult = 'adult' in request.POST.getlist('is_adult')
```

### 11.11.9 Dropdown for lookup

```
<select name="owner_id">
 <option value="0">Select an owner</option>
 {% for person in persons %}
 <option value={{person.person_id}}
 {% if vehicle.owner.person_id == person.person_id %} selected {% endif %}>
 {{ person.name_surname }}
 </option>
 {% endfor %}
</select>
```

In view:

```
owner_id = request.POST['owner_id']
try:
 owner = Person.objects.get(person_id = owner_id)
 vehicle.owner = owner
except:
 vehicle.owner = None
```

### 11.11.10 File

```
<!-- Display file name with link to open -->
<!-- Remember to map MEDIA_ROOT to MEDIA_URL in urls.py -->
{% if person.cv_file %}
 {{ person.cv_file_name_only }}
{% endif %}
<!-- Upload file -->
<input type="file" name="cv_file">
```

In view:

```
try:
 file = request.FILES['cv_file']
 if person.cv_file.name: #Remove existing file
 f_path = os.path.join(settings.MEDIA_ROOT, person.cv_file.name)
```

```

 if os.path.isfile(f_path): os.remove(f_path)
 person.cv_file = file
except: pass

```

### 11.11.11 Image file

```

<!-- Display file name -->
{% if person.photo_file %}
 {{person.photo_file_name_only}}

{% endif %}
<!-- Upload image -->
<input type="file" name="photo_file">
<!-- Show image -->
{% if person.photo_file %}
 <td><img src='{{person.photo_file.url}}' alt='{{person.photo_file.url}}'
height='300'></td>
{% endif %}

```

In view:

```

try:
 file = request.FILES['photo_file']
 if person.photo_file.name: #Remove existing file
 f_path = os.path.join(settings.MEDIA_ROOT, person.photo_file.name)
 if os.path.isfile(f_path): os.remove(f_path)
 person.photo_file = file
except: pass

```

## 11.12 Helpful procedures

### 11.12.1 Add & Edit with a single template

The add and edit templates are mostly very similar and a lot of repetition occurs. To save some work, try this:

- Develop the template as if it is for edit – include name and value attributes for all the input elements.
- Include a hidden element with the primary key as value attribute.
- For edit, pass `item_id` as context. For add, pass 0 as context.
- In the view, there is a single post for both add and edit and default value for the parameter.
- In `urls.py`, there are separate entries for the post function – one with a parameter (case for edit) and one without (case for add). Both entries point to the same view.

See the templates and views for person for an example.

### 11.12.2 Generic confirm delete

Instead of a separate delete html for every entity, develop a generic message:

`..templates\home\confirm.html:`

```

{% extends "base.html" %}

{% block content %}

<h1>Confirm</h1>

<form action="{{response_path_on_yes}}" method="post">
 {% csrf_token %}

 <p>{{message|linebreaks}}</p>

 <div align="center">
 <input type="submit" value=" Yes ">
 <input type="button" value=" No " onclick="window.location='{{response_path_on_no}}'">
 </div>
</form>

```

```

 </div>
 <p></p>
</form>

{% endblock %}

```

In the view:

```

def person_delete(request, person_id):
 person = Person.objects.get(person_id=person_id)
 return confirm(request,
 "Delete " + person.name_surname + "?", #msg
 reverse(person_delete_post, kwargs={'person_id': person_id}), #Response on 'Yes'
 reverse(person_list) #Response path on 'No'
)

```

### 11.12.3 Modal dialog boxes

Refer to the section on static files in Chapter 2 first (Section 2.4.4).

<https://raventools.com/blog/create-a-modal-dialog-using-css-and-javascript/>  
<https://gabrito.com/files/subModal/>  
[https://www.w3schools.com/howto/tryit.asp?filename=tryhow\\_css\\_modal2](https://www.w3schools.com/howto/tryit.asp?filename=tryhow_css_modal2)

## 11.13 Optional packages

For information on the user on client-side  
 pip install django\_user\_agents

For ImageField  
 pip install pillow

For getting client IP and geo-coding based on IP:  
 pip install requests

For geo-coding based on GPS  
 pip install geopy

## 11.14 Filters in views and templates

### 11.14.1 Basic

Operation	View	Template
Equals		{% if value == 0 %} Spaces around == important
Not equals	Section 5.2.5	{% if value != 0 %}
Larger than	Section 5.2.5	{% if value > 0 %}
AND	Model.objects.filter(x=1, y=2) Section 5.2.6	{% if x>2 and y < 3 %}
OR	Use   ( <a href="https://www.w3schools.com/django/django_queryset_filter.php">https://www.w3schools.com/django/django_queryset_filter.php</a> ) user_list = User.objects.filter(...) user_list  = User.objects.filter(...) Rather use __in (Section 5.2.6)	
NOT		{% if not condtion %}
Check for null	object__isnull=True	

[https://www.w3schools.com/django/django\\_queryset\\_filter.php](https://www.w3schools.com/django/django_queryset_filter.php)

### 11.14.2 Foreign keys: One-to-One

Consider the models User (built-in) and Custom\_User:

```
from django.contrib.auth.models import User
class Custom_User(models.Model):
 user = models.OneToOneField(User, models.DO_NOTHING, db_column='user_id',
 related_name="cu") #The last parameter is optional but preferred
 user_type = models.CharField(max_length=150, blank=True, null=True)

 class Meta:
 managed = True
 db_table = 'custom_user'
```

Forward querying: Starts from the secondary side and include all primary records.  
The parameter refers to the field name of the primary table inside the secondary table.  
cu\_list = Custom\_User.objects.select\_related('user').all()

Reverse querying: Starts from the primary side and include all secondary records -  
including records for which no secondary record exists.  
- If related name is not defined, the parameter refers to the db\_table name followed by \_set.  
user\_list = User.objects.prefetch\_related('custom\_user\_set').all()  
- If related\_name is defined:  
user\_list = User.objects.prefetch\_related('cu').all()

In the template, refer to the members like this:

```
{% for user in user_list %}
 user.cu.user_type
{% endfor %}
```

Alternatively, if you did not define related\_name:  
user.custom\_user.user\_type

Actually, we do not have to do a prefetch\_related.

In the view:

```
user_list = User.objects.all()
```

This will implicitly include all secondary records.

In the template the same as before:

```
user.cu.user_type
```

### 11.14.3 Foreign keys: One-to-Many

In case we can have more than one custom user object for every user (unlikely, but OK for the example):

```
from django.contrib.auth.models import User
class Custom_User(models.Model):
 user = models.ForeignKey(User, models.DO_NOTHING, db_column='user_id',
 related_name="cu") #The last parameter is optional but preferred
 user_type = models.CharField(max_length=150, blank=True, null=True)

 class Meta:
 managed = True
 db_table = 'custom_user'
```

The only difference is with respect to how we access the data in the template:

```
{% for user in user_list %}
 user.cu.all.0.user_type
```

```
{% endfor %}
```

Alternatively, if you did not define `related_name`:

```
user.custom_user_set.all().user_type
```

#### References

- [https://docs.djangoproject.com/en/5.0/topics/db/examples/one\\_to\\_one/](https://docs.djangoproject.com/en/5.0/topics/db/examples/one_to_one/)
- <https://vegibit.com/how-to-filter-foreign-key-in-django/>
- <https://docs.djangoproject.com/en/5.0/ref/models/queries/#prefetch-related>
- <https://stackoverflow.com/questions/9176430/django-does-prefetch-related-follow-reverse-relationship-lookup>

## 11.15 Typical error messages

### Browser errors

- `MultiValueDictKeyError`
  - This occurs after `request.POST['key']` in the view. Check the name attribute of the element in the corresponding template.
  - If the element refers to a file or image, check also `enctype` in the form.
- Page not found
  - Check the value for Request URL and check that it is available in `urls.py`.
  - Check that the parameters are correct.
  - Check that the syntax of the path in `urls.py` is correct (no extra or missing `'/'`, `'<'`, `'>'`, etc.).
- `TemplateDoesNotExist` at `/xxx/`
  - In `settings.py`, check that the template folder is included under `TEMPLATES`.
- Unable to connect
  - Check that the server is running. Check the server output for error messages.

### Compile errors

- `ImportError: cannot import name 'xxx' from 'app_xxx.models'`
  - Check all view files for an illegal import
- `ImportError: cannot import name 'xxx' from 'app_xxx.views.vw_xxx'`
  - Check that the function is defined in the view file.
- `ImportError: cannot import name 'vw_xxx' from 'app_xxx.views' (unknown location)`
  - Check that the view file has a proper extension.
  - You might have to close the terminal with running server and restart the server.
- `AttributeError: module 'app_pl3.views.vw_repository' has no attribute 'node_list'`
  - Check in `urls.py` for illegal imports
- `NameError: name 'vw_xxx' is not defined`
  - Check in `urls.py` for invalid references
- No module named `xxx`:  
Check the form's `enctype`

- Check that file names have the proper extension. No missing .html or .py extensions.

### Database

- not all arguments converted during bytes formatting
  - Check the number of %s parameters against the number of values in the sql string.
- (1136, "Column count doesn't match value count at row 1")
  - Check the number of %s parameters against the number of values in the sql string.

### Error log

- Time in the error log two hours behind:  
In settings.py. TIME\_ZONE = "Africa.Johannesburg"

### Files and images

- Files and images are opened when DEBUG=True, but not when DEBUG=False.
  - Don't set to False on localhost
  - On PythonAnywhere, on the web tab, under Static files:  
/media/ => /home/<username>/pr\_xxx/app\_xxx/media/
- Files and images don't open.
  - Check the media mapping in settings.py.
  - Check the media mapping at the end of urls.py.

### System errors

- System very slow
  - localhost: Check that VPN is off
  - remote: Check that remote database server and web server is OK

## 11.16 Sub-domains

A project (site) can have multiple apps, each pointing to a different sub-domain.

- In the command window:

```
..\env_replex\scripts> activate ← NB!!
..\env_replex> cd pr_replex
..\env_replex\pr_replex> py manage.py startapp app_sub
```

- Install Django-hosts:

```
..\env_replex\pr_replex> pip install django-hosts
```

- In VSCode, set up the folders and basic files as for the main app. Prefix file names with sub\_, e.g. sub\_index.html and sub\_home.py.

- In settings.py, add the new app under INSTALLED\_APPS. Add also Django-hosts:

```
INSTALLED_APPS = [
 ...
 'app_replex.apps.AppReplexConfig',
 'app_sub.apps.AppSubConfig',
 'django_hosts',
]
```

- In settings.py, under MIDDLEWARE, add the following at the beginning and the end respectively:

```
MIDDLEWARE = [
 'django_hosts.middleware.HostsRequestMiddleware',
 ...
 ...
 'django_hosts.middleware.HostsResponseMiddleware',
]
```

- In the app\_sub folder, create a separate urls.py.

```
from django.urls import path
from app_kosie.views import kosie_home

urlpatterns = [
 path('', kosie_home.index),
] #urlpatterns
```

- In the inner project folder (same folder as settings.py), add a file hosts.py. This file will serve to point the urls to the applicable domain or sub-domain views.

```
from django_hosts import patterns, host

host_patterns = patterns(
 '',
 host('', 'pr_replex.urls', name = ' '), #Points to urls.py under pr_
 host('sub', 'app_sub.urls', name='sub'), #Points to urls.py under app_sub
)
```

Note that the name parameter is compulsory.

- In settings.py, add the entries for the hosts:

```
ROOT_HOSTCONF = 'pr_replex.hosts'
DEFAULT_HOST = ' '
```

- For localhost, edit hosts:

```
C:\Windows\System32\drivers\etc\hosts
```



```
localhost name resolution is handled within DNS itself.
127.0.0.1 localhost
::1 localhost
127.0.0.1 sub.localhost
```

Perhaps not necessary?

Reference:

- <https://ordinarycoders.com/blog/article/django-subdomains>

## Chapter 12: Appendix E. PythonAnywhere

### 12.1 Development on PythonAnywhere

Instead of developing on localhost, you can optionally develop online on PythonAnywhere. Development and debugging are slower and there are no IntelliSense, but the advantage is that your app is available online immediately.

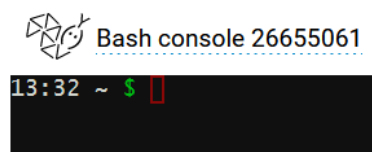
#### 12.1.1 PythonAnywhere account

- Create a free beginner account on [www.pythonanywhere.com](http://www.pythonanywhere.com).
- If you want me to give feedback:
  - Make sure that the username starts with your surname so that I will not be able to identify you on my side. Valid usernames would be: mofokengOrders, mofokengProjectPart1, mofokengTest1, etc.
  - Under the Account/Education tab, enter my username as your teacher: pieterblignaut (lower case, no spaces).
- Limitations of a free account:
  - You can register only one website at a time and you will have to create separate accounts for separate projects.
  - You cannot access a database from an external DBMS such as Navicat or DBeaver.
  - You cannot clear the error log manually with the echo command
- For more details to create an account on PythonAnywhere, see the separate document, [Create account](#).

#### 12.1.2 Bash console

The Bash console window serves as your command window in PythonAnywhere.

To open a Bash console window, go to the Consoles tab and click on Bash. It should show the local time and a prompt:



A piece of useless information: Bash stands for Bourne Again SHell - named after the developer, Bourne, who replaced the original Linux shell as developed by Thompson (<https://www.educative.io/courses/master-the-bash-shell/3j8399P3M6M>).

To clear the console:  
`$ clear`

#### 12.1.3 Virtual environment

Note: The project folder is outside the venv, but all its executables are inside the venv folder.

In the Bash console, type:  
`$ mkvirtualenv env_clothing`

Refer again to the naming conventions that was discussed in Section 1.5.1.

The prompt should now look like this:

```
(env_clothing) HH:mm ~ $
```

In future, we will just write:

```
$
```

Hint:

- If you want to start all over again,
  - remove the venv with

```
$ deactivate
```

```
$ rmvirtualenv env_clothing
```
  - Go to the web tab, scroll to the bottom and click on Delete
  - Go to the Files tab and delete all folders and files that you have added.

References

- <https://docs.python.org/3/tutorial/venv.html>
- <https://realpython.com/python-virtual-environments-a-primer/#what-is-a-python-virtual-environment>
- <https://latisresearch.umn.edu/python-virtual-environments-in-windows>
- <https://docs.djangoproject.com/en/4.1/intro/tutorial01/>
- <https://help.pythonanywhere.com/pages/FollowingTheDjangoTutorial/>
- <https://www.w3schools.com/django/>

#### 12.1.4 Install Django in the venv

```
(env_clothing) HH:mm ~ $ pip install django==4.1.7
```

NB: Specification of the version of Django is essential as later versions are not compatible with the version of MySQL that is used on PythonAnywhere.

Check version:

```
$ django-admin --version //Should say 4.1.7 or something similar
or
$ python -m django --version
```

#### 12.1.5 Activate the venv

If the prompt does not show the virtual environment, activate it:

```
$ workon env_clothing
```

Hint:

- Use deactivate to deactivate a current virtual environment.

#### 12.1.6 List all venvs

```
$ lsvirtualenv -b
```

#### 12.1.7 Create a project inside the venv

- Create the project

```
$ django-admin startproject pr_clothing
```
- Change folder

```
$ cd pr_clothing //Case sensitive!!
```

The prompt should now show the virtual environment in brackets and project as the current folder:

```
(env_clothing) HH:mm ~/pr_clothing $
```

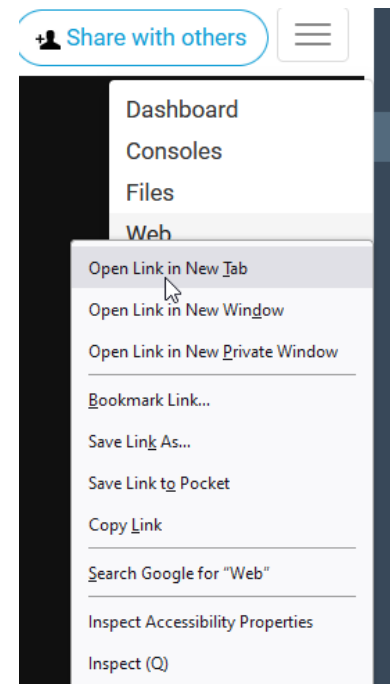
- NB: Most commands are executed from the project folder.

### 12.1.8 Create the web app

<https://docs.djangoproject.com/en/4.1/intro/tutorial01/>

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

- From the menu on the right-side of the screen, open the Web tab in a new tab in your browser window.



#### 12.1.8.1 Create App in PythonAnywhere

- Under Web tab, click "Add a new web app"
  - Specify manual configuration (NB: Not Django)
  - Python 3.10
  - Next
- Under the Virtualenv header
  - Enter env\_clothing (or whatever the venv name is)
  - It should change to /home/<username>/virtualenvs/env\_clothing  
Replace <username> with your personal username on PythonAnywhere. For me, it is /home/pieterblignaut/virtualenvs/env\_clothing
  - Reload web app (Green button at the top)


From <https://docs.djangoproject.com/en/4.1/intro/tutorial01/> :

*What's the difference between a project and an app? An app is a web application that does something – e.g., a blog system, a database of public records or a small poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.*

### 12.1.8.2 Update WSGI

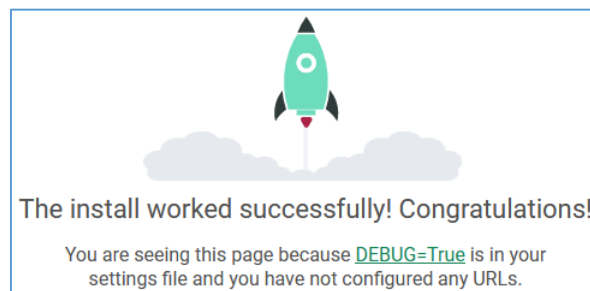
- Web tab
  - Under the header Code, click on the link for WSGI configuration file
  - Replace all with

```
import os
import sys

path = os.path.expanduser('~'/pr_clothing')
if path not in sys.path:
 sys.path.insert(0, path)
os.environ['DJANGO_SETTINGS_MODULE'] = 'pr_clothing.settings'
from django.core.wsgi import get_wsgi_application
from django.contrib.staticfiles.handlers import StaticFilesHandler
application = StaticFilesHandler(get_wsgi_application())
```
  - Note, if you copy and paste from here, make sure that you delete all extra leading spaces/tabs.
  - Make sure to replace `pr_clothing` with whatever the project name is.
  - Save (Green button top-right or press Ctrl-S)
- Maybe, check in the Files tab:
  - `../pr_clothing/pr_clothing/wsgi.py`, replace content as above. Save.
  - Save
  - Reload Web app (  top-right)

### 12.1.8.3 Test

Open a new tab in your browser and type: `<username>.pythonanywhere.com`. Of course, replace `<username>` with your username (defined in Section 12.1.1). You should see the rocket taking off as for the localhost.



If it does not work as expected, open the error log and see if you get some help there:

- From the dashboard, go to the web tab and click on the error log link.
- The log is cleared every day, but if you want to clear it manually, type this in the Bash console:

```
$ echo '' > /var/log/<username>.pythonanywhere.com.error.log
```

(Only available with a paid account.)

### 12.1.9 Create app in the Django project

```
~/pr_replex> python manage.py startapp app_replex
```

In the `app_replex` folder, create a new folder, `templates`.

```
(env_repex) ~\pr_repex $ cd app_repex
(env_repex) ~\pr_repex\app_repex $ mkdir templates
```

### 12.1.10 Update settings

- In ../pr\_clothing/pr\_clothing/Settings.py:  
  
ALLOWED\_HOSTS = ['\*']  
or  
ALLOWED\_HOSTS = ['<username>.pythonanywhere.com']
- The username has been defined in Section 12.1.1.

### 12.1.11 Add pages

- Use the Bash console to create a new folder, templates, in the app\_orders folder:  
(env\_orders) ~\pr\_orders \$ cd app\_orders  
(env\_orders) ~\pr\_orders\app\_orders \$ mkdir templates
- Add the templates, views and paths as for localhost.
- Save all files
- Rebuild web app 
- Test:
  - <username>.pythonanywhere.com

### 12.1.12 SSH

If you have a paid account on PythonAnywhere, you can access and edit your files on PythonAnywhere through VS Code. This gives you IntelliSense and a colourful coding environment.

On Windows:

- Settings / Apps / Optional features
  - Check that both OpenSSH Client and SSH Server is listed.
  - At the top, click “Add an optional feature” if necessary and follow the steps.
- Services / OpenSSH SSH Server
  - Startup type: Automatic
  - Start the service

In VS Code:

- View / Extensions
  - Install extension for Remote - SSH
  - Install
- Command palette
  - Press F1 or Ctrl-Shift-P
  - Remote-SSH: Add New SSH Host...
  - Platform: Linux
  - <username>@ssh.pythonanywhere.com
  - Enter your password on PythonAnywhere when prompted
- For the remote connection
  - Install extensions for Python and Django

References:

- <https://code.visualstudio.com/docs/remote/ssh>

## 12.2 Register a new domain

### 12.2.1 Create new web app

In PythonAnywhere

- Create a new venv, project and app as explained above.
- Under the Web tab
  - Note the CNAME url, for example `webapp-1979xxx.pythonanywhere.com`.
  - Under the Security heading, check Auto-renewed/Save and enable Force HTTPS.

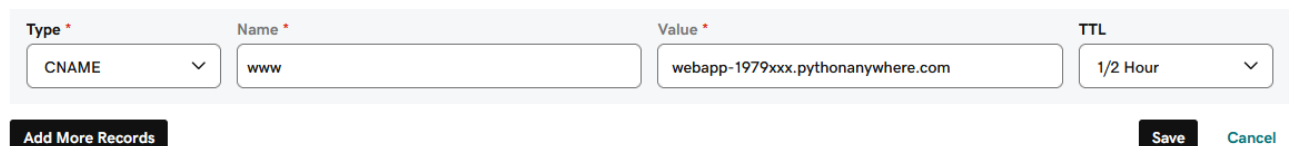
### 12.2.2 Register domain

- Create an account on GoDaddy and sign in: <https://sso.godaddy.com/>
- Go to shop and check if the required domain is available:  
<https://www.godaddy.com/en-ph>
- For the example, we will refer to [domain.co.za](https://www.godaddy.com/en-ph).
- Skip the website editing and publishing.







### 12.2.3 DNS setup

In GoDaddy:

- Under My Products, select the new domain.
- On the left, click on Domain, then Manage DNS. There should be no Type A records. There should be two NS (@ /ns41, @/ns42), CNAME (\_domainconnect) and one SOA (@/ns41) records. Delete all other records.
- ~~Click on add New Record and enter the CNAME url from PythonAnywhere.~~
- Edit the existing CNAME www record to point to the PythonAnywhere url.



- Add another CNAME record, with @ under Name and the same PythonAnywhere url under Value. These two records allow your user to type either <http://domain.co.za> and <http://www.domain.co.za> in a browser window.

	Type ①	Name ②	Data ③	TTL ④	Delete	Edit
<input type="checkbox"/>	A	@	15.197.142.173	600 seconds	Can't delete	Can't edit
<input type="checkbox"/>	A	@	3.33.152.147	600 seconds	Can't delete	Can't edit
<input type="checkbox"/>	NS	@	ns31.domaincontrol.com.	1 Hour	Can't delete	Can't edit
<input type="checkbox"/>	NS	@	ns32.domaincontrol.com.	1 Hour	Can't delete	Can't edit
<input type="checkbox"/>	CNAME	www	webapp-1971303.pythonanywhere.com.	1 Hour		
<input type="checkbox"/>	CNAME	_domainconnect	_domainconnect.gd.domaincontrol.com.	1 Hour		
<input type="checkbox"/>	SOA	@	Primary nameserver: ns31.domaincontrol.com.	1 Hour		

- If the naked domain above did not succeed, add a forwarding record for it:
  - Under DNS/Forwarding, click on Add Forwarding.

The image shows two side-by-side screenshots of the Godaddy 'Forwarding' configuration page. Both pages show the domain name at the top, a dropdown menu for the protocol (http:// or https://), a text input for the target URL, and three radio button options for the 'Forward Type': 'Permanent (301)' (selected), 'Temporary (302)', and 'Forward with masking'.

Left Screenshot: Domain is 'fouchewoodworking.co.za'. Protocol is 'http://'. Target URL is 'www.domain.co.za'. Forward Type is 'Permanent (301)'.

Right Screenshot: Domain is 'fileview.co.za'. Protocol is 'https://'. Target URL is 'www.fileview.co.za'. Forward Type is 'Permanent (301)'.

- Click on Save.
- Don't worry about the A records. They will be updated automatically after some time to point to the IP address of PythonAnywhere.
- For help
  - <https://ph.godaddy.com/help/contact-us>. Click on Chat now.
  - <https://help.pythonanywhere.com/pages/CustomDomains>
  - <https://ca.godaddy.com/help/add-a-cname-record-19236>

## 12.3 Publish on PythonAnywhere

- Create the environment, project, and database separately and from scratch on both localhost and PythonAnywhere. The database connection properties in the file settings.py will be different on localhost and PythonAnywhere.
- Do all development on localhost. Use VS Code and a DBMS. Test on localhost:8000.
- Once you are satisfied that the site is running well on localhost:8000, transfer all files to PythonAnywhere.
  - If you have a paid account on PythonAnywhere you can set up ssh on an ftp client, e.g. FileZilla, and copy all the files to PythonAnywhere. See the previous section for details about ssh.
  - If you have a free account, go to the Files tab and use the Upload facility of PA to upload files from localhost. Alternatively, you can create empty files with the same names on PA and then copy and paste the file content from your localhost editor (VS Code) to PA.
  - NB: Make sure that you do not overwrite the following files: `settings.py`, `wsgi.py`. When overwriting, make sure that you copy to the corresponding folder. You should only overwrite.
    - All templates (\*.html)
    - `forms.py`, `middleware.py`, `models.py`, `urls.py`
    - `views.py`
    - media folder
    - static folder
    - Database: Inspect the DDL on localhost and run the queries on PA.
- Reload the web app on PA.
- Check that all works as expected.

### 12.3.1 Install MySQL



The MySQL client must be installed for the virtual environment. In the Bash console:

```
$ workon env_clothing
$ cd pr_clothing
$ pip install mysqlclient
```

Create a database:

- Go to the Databases tab
- Initialise a MySQL database
  - Password: SOD517C\_ (minimum 8 characters)  
Please use this password so that I can access your databases.
  - Note the settings
  - Database name: clothing  
(Will be registered as <username>\$clothing, e.g. blignautClothing\$clothing).

If you want to remove a database, in the MySQL console:

```
$ drop database <username>$SOD;
```

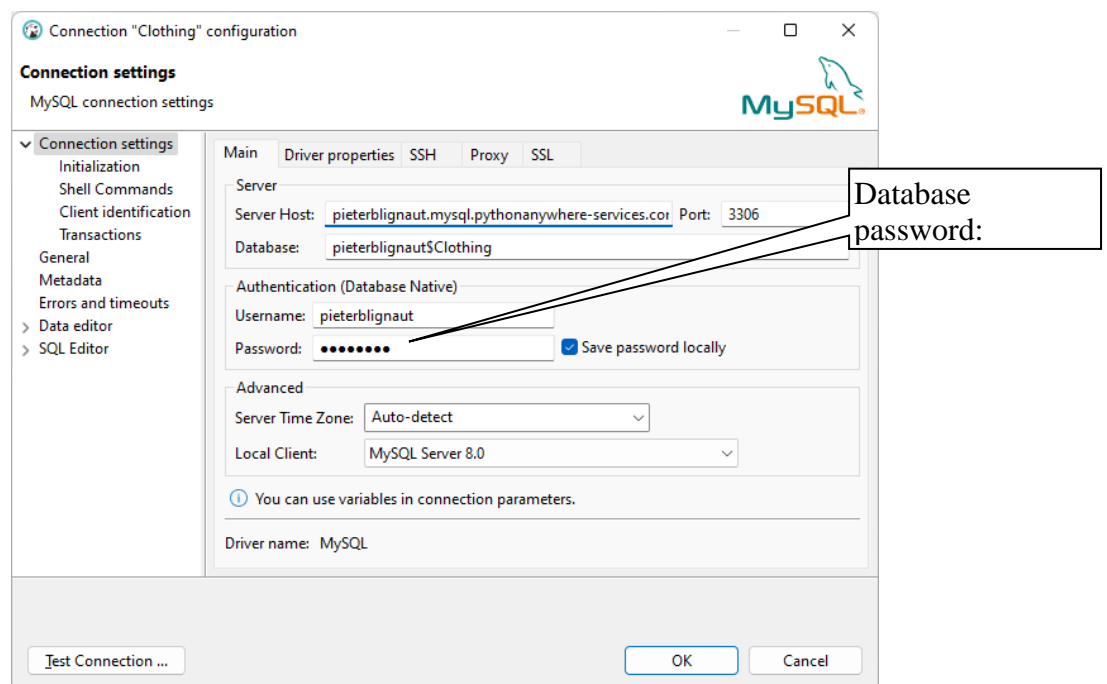
### 12.3.2 Install dependencies

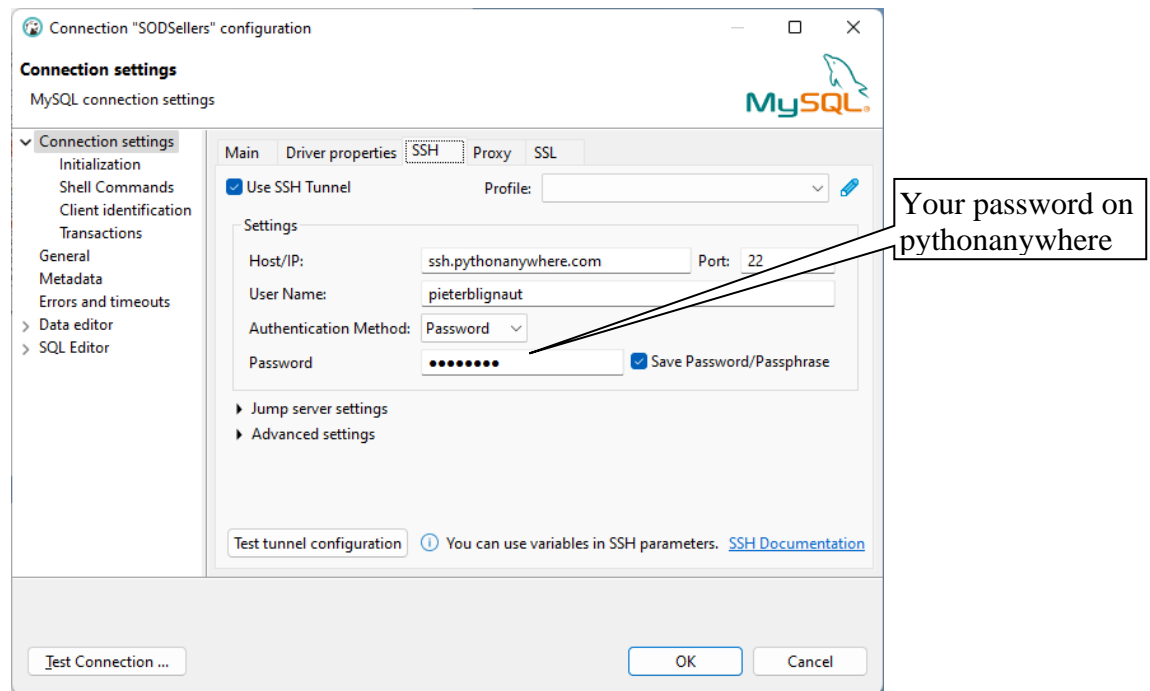
Install other dependencies that are specific for your application, for example pillow.

### 12.3.3 External connection to paid account

If you have a paid account, you can connect directly to the database in PythonAnywhere.

- Database / New Database Connection  
Select MySQL and click Next.
- Further instructions here:  
<https://help.pythonanywhere.com/pages/AccessingMySQLFromOutsidePythonAnywhere/>





### 12.3.4 Transfer the database to PythonAnywhere

If you have not already done so, create the database in PythonAnywhere (see Section **Error! Reference source not found.**).

- In the MySQL console (From the dashboard, under New console, click More and then MySQL)
  - Check that the database is available:  

```
mysql> show databases;
```

 You should see three databases, namely `information_schema`, `<username>$default` and `<username>$clothing`.
  - Ensure that you are not working with the default database:  

```
mysql> use <username>$clothing;
```
  - Note that all commands in the MySQL console are terminated with ;

Now, transfer the structure from localhost to PythonAnywhere:

- Working from left to right in the ERD, for each table in the database:
  - In DBeaver, right click on each table, Generate SQL / DDL. Copy the entire CREATE TABLE command.
  - In the MySQL console on PythonAnywhere, paste the command and press ENTER.
- In the MySQL console, check that all tables were copied  

```
mysql> show tables;
```

 You should see all the tables in the database.
- You can inspect the table definitions in the PythonAnywhere database with  

```
mysql> describe <tablename>;
```

Finally, we can transfer the test data to the PythonAnywhere database:

- Again, working from left to right in the ERD, for each table in the database:
  - In DBeaver, double click on the table name in the left-hand panel.

- Under the Data tab on the right-hand panel, right-click and select Export data.
- Select SQL.
- Proceed through the steps of the wizard.
  - Format settings: Omit schema name ([v])
  - Output: "Copy to clipboard".
- Click Proceed on the last step.
- In MySQL console on PythonAnywhere, paste the command and press ENTER.

### 12.3.5 Map the static urls

On the web tab, under Static files, map the media url to the full path on disk:

```
/media/ => /home/<username>/pr_/app_/media/
```

In settings.py, set Debug= False.

### 12.3.6 Copy files with SSH

Use FileZilla or similar tool to FTP the documents

- Set up the connection
  - Host: ssh.pythonanywhere.com
  - Username: Your username on PA.
  - Password: Your password on PythonAnywhere
  - Port: 22
- Edit settings.py manually. It is not identical to the version on localhost.
  - DEBUG = False
  - APPEND\_SLASH = True
  - TEMPLATES / DIRS
  - DATABASES  
(MySQL instead of SQLite)
- Copy files (excluding settings.py) to the corresponding folder
  - env\_/pr\_/pr\_
    - urls.py
    - Don't copy asgi.py, settings.py or wsgi.py.
  - env\_/pr\_/app\_
    - models.py
    - /templates
    - /views
    - /forms, /middleware (If any)
    - /static, /media/files, /media/images  
(Maybe not. Rather upload the files from the published version of your app.)
- Migrate
  - If models.py is copied, there is no need to create the database explicitly.
  - This must be done every time that models.py is updated.
  - In the Bash console:
 

```
(env_XXX) ~/pr_XXX $ python manage.py makemigrations
(env_XXX) ~/pr_XXX $ python manage.py migrate
```
- Reload the website
  - This must be done every time that a view or template is updated.

## Chapter 13: Appendix F (Source code on the running example)

### 13.1 urls.py

```
from django.urls.conf import path, re_path
from django.contrib import admin

from app_clothing.views.vw_home import index
from app_clothing.views.vw_admin import sql, sql_post, usage_log
from app_clothing.views.vw_files import files, files_post, file_open, file_delete, file_delete_post
from app_clothing.views.vw_customers import customers, customer_list, customer_add, customer_add_post,
customer_edit, customer_edit_post, customer_delete, customer_delete_post
from app_clothing.views.vw_products import product_list_1, product_list_2, product_list_3, \
product_list_3b, product_list_4, product_details, \
product_add, product_add_post, product_edit,
product_edit_post, product_delete, product_delete_post
from app_clothing.views.vw_suppliers import supplier_list, supplier_add, supplier_add_post, \
supplier_edit, supplier_edit_post, supplier_delete,
supplier_delete_post
from app_clothing.views.vw_users import user_list, user_login, user_login_post, user_logout, \
user_password, user_password_post
from app_clothing.views.vw_email import email, email_post
from app_clothing.views.serializers_book import BookList, BookDetails
from app_clothing.views.serializers_supplier import SupplierList, SupplierDetails

urlpatterns = [
 #Admin
 path('admin/', admin.site.urls),

 #Index
 path('', index, name='index'),
 path('index/', index),

 #Admin
 path('sql/', sql),
 path('sql_post/', sql_post),
 path('usage_log/', usage_log),

 #Customers
 path('customers/', customers),
 path('customer_list/', customer_list),
 path('customer_add/', customer_add),
 path('customer_add_post/', customer_add_post),
 path('customer_edit/<int:customer_id>/', customer_edit),
 path('customer_edit_post/<int:customer_id>/', customer_edit_post),
 path('customer_delete/<int:customer_id>/', customer_delete),
 path('customer_delete_post/<int:customer_id>/', customer_delete_post),

 #Products
 path('product_list/', product_list_3),
 path('product_list/<int:supplier_id>/', product_list_3b),

 path('product_details/<str:product_code>/', product_details),
 path('product_add/', product_add),
 path('product_add_post/', product_add_post),
 path('product_edit/<int:product_id>/', product_edit),
 path('product_edit_post/<int:product_id>/', product_edit_post),
 path('product_delete/<int:product_id>/', product_delete),
 path('product_delete_post/<int:product_id>/', product_delete_post),

 #Suppliers
 path('supplier_list/', supplier_list),
 path('supplier_add/', supplier_add),
 path('supplier_add_post/', supplier_add_post),
 path('supplier_edit/<int:supplier_id>/', supplier_edit),
 path('supplier_edit_post/<int:supplier_id>/', supplier_edit_post),
 path('supplier_delete/<int:supplier_id>/', supplier_delete),
 path('supplier_delete_post/<int:supplier_id>/', supplier_delete_post),
```

```

#Users
path('user_list/', user_list),
path('user_login/', user_login),
path('user_logout/', user_logout),
path('user_login_post/', user_login_post),
path('user_password/<int:user_id>', user_password),
path('user_password_post/<int:user_id>', user_password_post),

#Email
path('email/<int:user_id>', email),
path('email_post/<int:user_id>', email_post),

#Files
path('files/', files),
path('files_post/', files_post),
path('file_open/<str:file_name>', file_open),
path('file_delete/<str:file_name>', file_delete),
path('file_delete_post/<str:file_name>', file_delete_post),

#Serializers
path('api_books/', BookList.as_view()),
path('api_books/<int:book_id>', BookDetails.as_view()),
path('api_suppliers/', SupplierList.as_view()),
path('api_suppliers/<int:supplier_id>', SupplierDetails.as_view()),

] #end urlpatterns

#For media
from django.conf import settings
from django.conf.urls.static import static
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

```

## 13.2 models.py

```

from django.db import models

class Book(models.Model):
 book_id = models.AutoField(primary_key=True)
 title = models.TextField(blank=True, null=True)
 author = models.TextField(blank=True, null=True)

 class Meta:
 managed = False
 db_table = 'Book'

class Customer(models.Model):
 customer_id = models.AutoField(primary_key=True)
 user = models.OneToOneField('AuthUser', models.DO_NOTHING)
 customer_name = models.TextField()
 address = models.TextField(blank=True, null=True)
 phone_number = models.TextField(blank=True, null=True)

 class Meta:
 managed = False
 db_table = 'Customer'

import os
class Product(models.Model):
 product_id = models.AutoField(primary_key=True)
 product_code = models.TextField(blank=True, null=True)
 description = models.TextField(blank=True, null=True)
 description_file = models.FileField(db_column="description_file_name", upload_to='files/')
 image_file = models.ImageField(db_column="image_file_name", upload_to='images/')
 price = models.DecimalField(max_digits=10, decimal_places=5, blank=True, null=True)
 supplier = models.ForeignKey('Supplier', models.DO_NOTHING, blank=True, null=True)

 class Meta:
 managed = False
 db_table = 'Product'

 def description_file_name_only(self):
 return os.path.basename(self.description_file.name)
 def image_file_name_only(self):
 return os.path.basename(self.image_file.name)

```

```

class Sale(models.Model):
 sale_id = models.AutoField(primary_key=True)
 customer = models.ForeignKey(Customer, models.DO_NOTHING, blank=True, null=True)
 product = models.ForeignKey(Product, models.DO_NOTHING, blank=True, null=True)
 invoice_number = models.TextField(blank=True, null=True)

 class Meta:
 managed = False
 db_table = 'Sale'

class Supplier(models.Model):
 supplier_id = models.AutoField(primary_key=True)
 user = models.OneToOneField('AuthUser', models.DO_NOTHING) #, null=True, blank=True)
 supplier_name = models.TextField(blank=True, null=True)
 address = models.TextField(blank=True, null=True)
 phone_number = models.TextField(blank=True, null=True)

 class Meta:
 managed = False
 db_table = 'Supplier'

```

## 13.3 Home

### 13.3.1 base.html

```

<!DOCTYPE html>
<html>

<head>
 <style>
 body { text-align: center;}
 .div_center { display:inline-block; border: 1px solid;}
 ul, li, table { text-align: left; }
 ul { padding-left:20px; }
 h2 { padding:0px 10px 0px 10px}
 </style>

 <!-- In the templates, surround the content also with <script></script> tags. If we put
 the tags here, we would lose the JavaScript intellisense in the templates. -->
 {% block scripts %}
 {% endblock scripts %}

</head>

<body>
 <div class="div_center">
 {% block content %}
 {% endblock content %}

 <!-- Messages -->
 <div style="text-align: left;">
 {% if messages %}
 <h3>Messages</h3>
 {% for message in messages %}
 {{ message }}
 </br>
 {% endfor %}
 {% endif %}
 </div>
 </div>
</body>
</html>

```

### 13.3.2 index.html

```

{% extends "base.html" %}

{% block content %}

 <h2>Clothing shop</h2>

 Log in

 {% if request.user.is_authenticated %}
 <p>User name: {{ request.user.username }}</p>

```

#### Clothing shop

- [Log in](#)

#### Clothing shop

- [Log in](#)
- User name: a
- [Customers \(raw\)](#)
- [Customers](#)
- [Suppliers](#)
- [Products](#)
- [Users](#)
- [Administration](#)
- [SQL](#)
- [Usage log](#)
- [Files](#)
- [Log out](#)

```

{% if perms.app_clothing.view_customer %}
 Customers (raw)
 Customers
{% endif %}
{% if perms.app_clothing.view_supplier %}
 Suppliers
{% endif %}

Products

{% if request.user.is_superuser %}

 Users

 Administration

 SQL
 Usage log

 Files

{% endif %}

 Log out
{% endif %}

{% endblock %}

```



### 13.3.3 message.html

```

<!DOCTYPE html>

<head>
 <style>
 body { text-align: center; }
 .div_left { display:inline-block; border: 1px solid; text-align: left; padding: 0px 20px 0px
20px;}
 h2 { padding:0px 10px 0px 10px}
 </style>
</head>

<html>
<body>

 <div class="div_left">
 <h2>MESSAGE</h2>
 <pre>{{ message | linebreaksbr }}</pre>
 <div align="center">
 <input type="button" value="OK" text-align="center" onclick="window.location='/'>
 <p></p>
 </div>
 </div>

</body>

</html>

```

### 13.3.4 vw\_home.py

```
from django.shortcuts import render

def index(request):
 return render(request, 'index.html')

def show_message(request, message):
 return render (request, "message.html", { "message": message})
```



## 13.4 Admin

### 13.4.1 sql\_form.html

**SQL**

supplier_id	user_id	supplier_name	address	phone_number
2	9	Supplier 1	Address 1a Address 1b Address 1c	1234
3	10	Supplier 2	Address 2a Address 2b	2345
4	11	Supplier 3	Address 3a Address 3b Address 3c	3456
5	12	Supplier 4	Address 4a Address 4b	45
6	13	Supplier 5	Address 5a Address 5b	567
7	14	Supplier 6	Address 6a Address 6b Address 6c Address 6d	678
8	17	Supplier 7	Address 7a	789
10	2	Supplier 11	S11 address	S11 phone
11	3	Supplier 11	S11 address	S11 phone
12	4	Supplier 11	S11 address	S11 phone
14	5	S11	Address 9	99

```
{% extends "base.html" %}
{% block content %}

<form action="/sql_post/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>SQL</td>
 <td><input name="sql" size="50" value="{{sql}}"></td>
 </tr>
 <tr>
 <td>
</td>
 </tr>
 <tr>
 <td colspan="2" align="center">
 <input type="submit" value="Submit">
 <input type="button" value="Cancel" onclick="window.location='/';">
 </td>
 </tr>
 </table>

 {% if "SELECT" in sql %}
 <table>
 <!-- Headers -->
 <tr>
 {% for col in cols %}
 <td align="left">{{ col }}</td>
 {% endfor %}
 </tr>

 <!-- Data rows -->
 {% for row in rows %}
 <tr>
 {% for col in row %}
 <td>{{ col }}</td>
 {% endfor %}
 </tr>
 {% endfor %}
 </table>
 {% endif %}

 {% if error %}
 {{ error | linebreaksbr }}
 {% endif %}

</form>

{% endblock %}
```

### 13.4.2 usage\_log.html

## Usage log

User

Date from

Date to

Date and time	User name	Path
2023-12-22 11:32:27	a	/email_post/15/
2023-12-22 11:32:27	a	/favicon.ico
2023-12-22 11:32:28	a	/
2023-12-22 11:32:54	a	/user_login/
2023-12-22 11:34:00	a	/user_login_post/
2023-12-22 11:34:02	a	/user_list/
2023-12-22 11:34:04	a	/user_password/10/

```
{% extends "base.html" %}
{% block content %}
```

```
<h1>Usage log</h1>
<form action="/usage_log/" method="post">
 {% csrf_token %}

 <!-- Column headers -->
 <table>
 <tr>
 <td>User</td>
 <td>
 <select name="user_id">
 {% for user in user_list %}
 {% if user.id == selected_user_id %}
 <option value="{{ user.id }}" selected>
 {{ user.username }}
 </option>
 {% else %}
 <option value="{{ user.id }}">
 {{ user.username }}
 </option>
 {% endif %}
 {% endfor %}
 </select>
 </td>
 </tr>
 <tr>
 <td>Date from</td>
 <td><input type="date" name="date_from"
 value="{{selected_date_from}}"></td>
 </tr>
 <tr>
 <td>Date to</td>
 <td><input type="date" name="date_to"
 value="{{selected_date_to}}"></td>
 </tr>
 <tr>
 <td></td>
 <td><input type="submit" value="Submit"></td>
 </tr>

 <tr>
 <th width="200px" align="left">Date and time</th>
 <th width="120px" align="left">User name</th>
 <th width="200px" align="left">Path</th>
 </tr>
 </table>
```

```

<!-- Data rows -->
<table>
 {% for row in log %}
 <tr>
 <td width="200px">{{ row.date_time }}</td>
 <td width="120px">{{ row.username }}</td>
 <td width="200px">{{ row.path }}</td>
 </tr>
 {% endfor %}
</table>

<div align="center">
 <input type="button" value="Home" onclick="window.location='/'">
</div>
<p></p>
</form>
{% endblock %}

```

### 13.4.3 vw\_admin.py

```

from django.shortcuts import render
from . import vw_home
from django.contrib import messages
from django.db import connection
from django.contrib.auth.models import User
from datetime import date

def sql(request):
 #messages.info(request, "sql")
 return render(request, 'sql_form.html')

def sql_post(request):
 if request.method == "POST":
 sql = request.POST['sql']
 cursor = connection.cursor()
 try:
 #return vw_home.show_message(request, "3")
 cursor.execute(sql)
 if "SELECT" in sql:
 columns = [col[0] for col in cursor.description]
 rows = cursor.fetchall()
 return render(request, 'sql_form.html', {'sql': sql, 'cols': columns, 'rows': rows})
 else:
 return render(request, 'sql_form.html', {'sql': sql})
 except Exception as e:
 return render(request, 'sql_form.html', {'sql': sql, 'error': e})
 return vw_home.index(request)

```

```

def usage_log(request):
 user_list = User.objects.all();

 if request.method == 'POST':
 #Get user selections
 user_id = int(request.POST['user_id'])
 date_from = request.POST['date_from']
 date_to = request.POST['date_to']

 #Query DB
 cursor = connection.cursor()
 sql = """SELECT strftime(%s,date_time) AS date_time, username, path
 FROM log LEFT JOIN auth_user U ON log.user_id = U.id
 WHERE path <> ' ' """
 if (user_id > 0 and date_from != "" and date_to != ""):
 sql += """AND user_id = %s
 AND date_time >= %s
 AND date_time <= DATE(%s, '+1 days') """
 params = ['%Y-%m-%d %H:%M:%S', user_id, date_from, date_to]
 elif (user_id > 0):
 sql += "AND user_id = %s "
 params = ['%Y-%m-%d %H:%M:%S', user_id]
 elif (date_from != "" and date_to != ""):
 sql += """AND date_time >= %s
 AND date_time <= DATE(%s, '+1 days') """
 params = ['%Y-%m-%d %H:%M:%S', date_from, date_to]
 else:
 params = ['%Y-%m-%d %H:%M:%S']
 sql += "ORDER BY date_time"
 cursor.execute(sql, params)

 #Get data
 columns = [col[0] for col in cursor.description]
 rows = [dict(zip(columns, row)) for row in cursor.fetchall()]

 #Render html with log of selected user and date interval
 return render (request, "usage_log.html",
 {'user_list': user_list,
 'selected_user_id': user_id,
 'selected_date_from': date_from,
 'selected_date_to': date_to, 'log': rows})
 else:
 #Render html with initial filter values and empty log
 return render (request, "usage_log.html",
 {'user_list': user_list,
 'selected_user_id' : -1,
 'selected_date_from' : date.today().strftime('%Y-%m-%d'),
 'selected_date_to' : date.today().strftime('%Y-%m-%d') })

```

## 13.5 Users

### 13.5.1 user\_list.html

```
{% extends "base.html" %}
{% block content %}

<h1>Users</h1>

<!-- Column headers -->
<table>
 <tr>
 <th width="70px" align="left">User id</th>
 <th width="120px" align="left">User name</th>
 <td></td>
 </tr>
</table>

<!-- Data rows -->
<table>
 {% for user in user_list %}
 <tr>
 <td width="70px">{{ user.id }}</td>
 <td width="120px">{{ user.username }}</td>
 <td>Change password</td>
 </tr>
 {% endfor %}
</table>

<div align="center">

 <input type="button" value="Home" onclick="window.location='/'">
 <p></p>
</div>

{% endblock %}
```

Users		
User id	User name	
1	clothing	<a href="#">Change password</a>
2	a	<a href="#">Change password</a>
3	c	<a href="#">Change password</a>
19	c4	<a href="#">Change password</a>
20	c5	<a href="#">Change password</a>
<input type="button" value="Home"/>		

### 13.5.2 user\_login.html

```
{% extends "base.html" %}
{% block content %}

<form action="/user_login_post/" method="post">
 {% csrf_token %}

 <table>
 <tr>
 <td width="100%" colspan="2" align="center">
 <h1>Log in (custom)</h1>
 </td>
 </tr>
 <tr>
 <td>User name</td>
 <td><input name="username" onkeypress="return event.charCode != 32"></td>
 </tr>
 <tr>
 <td>Password</td>
 <td><input type="password" name="password" onkeypress="return event.charCode != 32"></td>
 </tr>
 <tr><td>
</td></tr>
 <tr>
 <td colspan="2" align="center">
 <input type="submit" value="Log in">
 </td>
 </tr>
 </table>
</form>

{% endblock %}
```

Log in (custom)	
User name	<input type="text" value="a"/>
Password	<input type="password"/>
<input type="button" value="Log in"/>	

### 13.5.3 user\_password.html

```
{% extends "base.html" %}
{% block content %}

<h1>Change user password</h1>

<form action="/user_password_post/{{ user.id }}"
method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Username</td>
 <td>{{ user.username }}</td>
 </tr>
 <tr>
 <td>Old password</td>
 <td><input type="password" name = "old_password"></td>
 </tr>
 <tr>
 <td>New password</td>
 <td><input type="password" name="new_password"></td>
 </tr>
 <tr>
 <td>New password again</td>
 <td><input type="password" name="new_password_again"></td>
 </tr>
 <tr></tr>
 <tr>
 <td></td>
 <td>
 <input type="submit" value="Submit">
 <input type="button" value="Cancel" onclick="window.location='/user_list/'">
 </td>
 </tr>
 </table>

</form>

{% endblock %}
```

Change user password

Username	clothing
Old password	<input type="password"/>
New password	<input type="password"/>
New password again	<input type="password"/>

Submit Cancel

### 13.5.4 vw\_users.py

```
from django.shortcuts import render
from django.contrib import messages
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.models import Group, Permission, User
from . import vw_home

#User list
def user_list(request):
 users = User.objects.all()
 return render (request, "user_list.html", {'user_list': users})

#Login/out
def user_login(request):
 return render(request, 'user_login.html')

def user_login_post(request):
 if request.method == 'POST':
 un = request.POST['username']
 pwd = request.POST['password']
 user = authenticate(username=un, password=pwd)
 if user is not None:
 login(request, user)
 #messages.info(request, "LoginPost")
 #Permissions(request)
 else:
 logout(request)
 messages.error(request, "Authentication failed")
 else:
 messages.error(request, "Not posted")
 return vw_home.index(request)
```

```

def user_logout(request):
 logout(request)
 return vw_home.index(request)

#Password
def user_password(request, user_id):
 user = User.objects.get(id=user_id)
 return render (request, "user_password.html", {'user': user})

def user_password_post(request, user_id):
 if request.method == 'POST':
 user = User.objects.get(id = user_id)
 old_password = request.POST['old_password'];
 new_password_1 = request.POST['new_password'];
 new_password_2 = request.POST['new_password_again'];
 if user.check_password(old_password) \
 and new_password_1 == new_password_2:
 user.set_password(new_password_1);
 user.save()
 return vw_home.show_message(request, "Password changed")
 else:
 return vw_home.show_message(request, "Invalid old password or new passwords do not
match.")
 return user_list(request)

```

## 13.6 Customers

### 13.6.1 customer\_list.html

CUSTOMERS					
ID	Name	Address		Telephone	<a href="#">Add customer</a>
1	Customer 1	Address 1a Address 1b	<a href="#">Email</a>	1234	<a href="#">Edit</a> <a href="#">Delete</a>
2	Customer 2	Address 2a Address 2b Address 2c	<a href="#">Email</a>	2345	<a href="#">Edit</a> <a href="#">Delete</a>
3	Customer 3	Address 3a Address 3b Address 3b	<a href="#">Email</a>	3456	<a href="#">Edit</a> <a href="#">Delete</a>
4	Customer 4	Address 4a Address 4b	<a href="#">Email</a>	4567	<a href="#">Edit</a> <a href="#">Delete</a>
5	Customer 5	Address 5a Address 5b	<a href="#">Email</a>	5678	<a href="#">Edit</a> <a href="#">Delete</a>

```
{% extends "base.html" %}
{% block content %}

 <h2>CUSTOMERS</h2>

 <!-- Column headers -->
 <table>
 <tr>
 <th width="70px" align="left">ID</th>
 <th width="120px" align="left">Name</th>
 <th width="200px" align="left">Address</th>
 <th width="80px" align="left"></th>
 <th width="120px" align="left">Telephone</th>
 {% if request.user.is_superuser %}
 <td>Add customer</td>
 {% endif %}
 <td></td>
 </tr>
 </table>

 <!-- Data rows -->
 <table>
 {% for customer in customer_set %}
 <tr valign="top">
 <td width="70px">{{ customer.customer_id }}</td>
 <td width="120px">{{ customer.customer_name }}</td>
 <td width="200px">{{ customer.address | linebreaksbr }}</td>
 <td width="80px">Email</td>
 <td width="120px">{{ customer.phone_number }}</td>
 {% if request.user.is_superuser %}
 <td>Edit</td>
 <td>Delete</td>
 {% else %}
 {% if perms.app_clothing.change_customer %}
 <td>Edit</td>
 {% endif %}
 {% endif %}
 </tr>
 {% endfor %}
 </table>

 <input type="button" value="Home" onclick="window.location='/'">
 <p></p>
{% endblock %}
```



### 13.6.2 customer\_add.html

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

{% block scripts %}
<script>
 function UpdateHidden()
 {
 //textarea does not have a name attribute
 address = document.getElementById('taAddress').value;
 document.getElementById('hiddenAddress').value = address;
 }
</script>
{% endblock %}

{% block content %}

 <h2>ADD CUSTOMER</h2>

 <form action="/customer_add_post/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Customer name</td>
 <td><input name="name"></td>
 </tr>
 <tr>
 <td>User name</td>
 <td><input name="username" onkeypress="return event.charCode != 32"></td>
 </tr>
 <tr>
 <td>Password</td>
 <td><input type="password" name="password"
 onkeypress="return event.charCode != 32"></td>
 </tr>
 <tr>
 <td>Address</td>
 <td>
 <textarea rows="5" cols="60" id="taAddress"
 onchange="UpdateHidden();"></textarea>
 <input type="hidden" id="hiddenAddress" name="address"></hidden>
 </td>
 </tr>
 <tr>
 <td>Email address</td>
 <td><input name="email"></td>
 </tr>
 <tr>
 <td>Phone number</td>
 <td><input name="phone"></td>
 </tr>
 <tr>
 <td colspan="2">
 <p></p>
 <input type="submit" value="Add">
 <input type="button" value="Cancel" onclick="window.location='/customer_list/'">
 <p></p>
 </td>
 </tr>
 </table>

 </form>

{% endblock %}
```



### 13.6.3 customer\_edit.html

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

{% block scripts %}
<script>
 function UpdateHidden()
 {
 //textarea does not have a name attribute
 address = document.getElementById('taAddress').value;
 document.getElementById('hiddenAddress').value = address;
 }
</script>
{% endblock %}

{% block content %}
```

```

 <h2>EDIT CUSTOMER</h2>

 <form action="/customer_edit_post/{{customer.customer_id}}/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Customer name</td>
 <td><input name="name" value="{{ customer.customer_name }}"></td>
 </tr>
 <tr>
 <td>Address</td>
 <td>
 <textarea rows="5" cols="60" id="taAddress"
 onchange="UpdateHidden();">{{customer.address}}</textarea>
 <input type="hidden" id="hiddenAddress" name="address"
 value="{{customer.address }}"></hidden>
 </td>
 </tr>
 <tr>
 <td>Email address</td>
 <td><input name="email" value="{{customer.user.email}}"></td>
 </tr>
 <tr>
 <td>Phone number</td>
 <td><input name="phone" value="{{customer.phone_number}}"></td>
 </tr>
 <tr></tr>
 <tr>
 <td colspan="2" align="center">
 <p></p>
 <input type="submit" value="Submit">
 <input type="button" value="Cancel" onclick="window.location='/customer_list/'">
 <p></p>
 </td>
 </tr>
 </table>

 </form>

{% endblock %}
```

### 13.6.4 customer\_delete.html

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

{% block content %}

<h2>DELETE CUSTOMER</h2>

<form action="/customer_delete_post/{{ customer.customer_id }}"
 method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete {{ customer.customer_name
 }}?</p>

 <div align="center">
 <input type="submit" class="submit-btn" value=" Yes ">
 <input type="button" class="submit-btn" value=" No "
 onclick="window.location='/customer_list/'">
 <p></p>
 </div>
</form>
{% endblock %}
```

**DELETE CUSTOMER**  
Are you sure you want to delete Customer 1?

### 13.6.5 vw\_customers.py

```
from django.shortcuts import HttpResponseRedirect, render
from app_clothing.models import Customer
from . import vw_home
from django.contrib.auth.models import Group, Permission, User

def customers(request):
 customers = Customer.objects.all().order_by('customer_name') #Find the data in models
 #output = ', '.join([c.customer_name for c in customers])
 output = ', '.join([str(c) for c in customers])
 return HttpResponseRedirect(output)

def customer_list(request):
 customer_set = []
 if request.user.is_superuser:
 customer_set = Customer.objects.order_by('customer_name')
 if request.user.groups.filter(name="Customers").exists():
 customer_set = Customer.objects.filter(user__id=request.user.id)

 context = {'customer_set': customer_set, }
 #return vw_home.show_message(request, "Supplier list")
 return render (request, "customer_list.html", context)

def customer_add(request):
 if request.user.is_superuser:
 return render (request, "customer_add.html")
 return vw_home.show_message(request, "Not permitted");

def customer_add_post(request):
 if request.method == 'POST':
 #Get values from the template
 username = request.POST['username']
 password = request.POST['password']
 name = request.POST['name']
 address = request.POST['address']
 email = request.POST['email']
 phone = request.POST['phone']

 #Create new user and add the user to the appropriate group
 user = User.objects.create_user(username=username, password=password, email=email)
 group = Group.objects.get(name="Customers")
 group.user_set.add(user)

 #Create new customer
 customer = Customer(customer_name=name, user_id=user.id, address= address, \
 phone_number = phone)
 customer.save()
 return customer_list(request)
```

```

def customer_edit(request, customer_id):
 customer = Customer.objects.get(customer_id = customer_id)
 if request.user.is_superuser or request.user.id == customer.user.id:
 return render (request, "customer_edit.html", {'customer': customer})
 return vw_home.show_message(request, "Not permitted");

def customer_edit_post(request, customer_id):
 if request.method == 'POST':
 customer = Customer.objects.get(customer_id=customer_id)
 customer.customer_name = request.POST['name']
 customer.address = request.POST['address']
 customer.phone_number = request.POST['phone']
 customer.save()
 customer.user.email = request.POST['email']
 customer.user.save();
 return customer_list(request)

def customer_delete(request, customer_id):
 if request.user.is_superuser:
 customer = Customer.objects.get(customer_id = customer_id)
 return render (request, "customer_delete.html", {'customer': customer})
 return vw_home.show_message(request, "Not permitted");

def customer_delete_post(request, customer_id):
 customer = Customer.objects.get(customer_id = customer_id)
 if request.method == 'POST':
 try:
 user_id = customer.user.id
 customer.delete()
 user = User.objects.get(id = user_id)
 user.delete()
 except:
 #return vw_home.show_message(request, e.__cause__)
 return vw_home.show_message(request, "Cannot delete customer '" \
 + customer.customer_name + "'.\nA related sale exists.")
 return customer_list(request)

```

## 13.7 Suppliers

### 13.7.1 supplier\_list.html

```
{% extends "base.html" %}
{% block content %}
```

```
<h2>SUPPLIERS</h2>
```

```
<!-- Column headers -->
```

```
<table>
```

```
<tr>
```

```
<th width="70px" align="left">ID</th>
```

```
<th width="120px" align="left">Name</th>
```

```
<th width="200px" align="left">Address</th>
```

```
<th width="80px" align="left"></th>
```

```
<th width="120px" align="left">Telephone</th>
```

```
{% if request.user.is_superuser %}
```

```
<td>Add supplier</td>
```

```
{% endif %}
```

```
<td></td>
```

```
</tr>
```

```
</table>
```

```
<!-- Data rows -->
```

```
<table>
```

```
{% for supplier in supplier_set %}
```

```
<tr valign="top">
```

```
<td width="70px">{{ supplier.supplier_id }}</td>
```

```
<td width="120px">{{ supplier.supplier_name }}</td>
```

```
<td width="200px">{{ supplier.address | linebreaksbr }}</td>
```

```
<td width="80px">Email</td>
```

```
<td width="120px">{{ supplier.phone_number }}</td>
```

```
{% if request.user.is_superuser %}
```

```
<td>Edit</td>
```

```
<td>Delete</td>
```

```
<td>Products</td>
```

```
{% else %}
```

```
{% if perms.app_clothing.change_supplier %}
```

```
<td>Edit</td>
```

```
<td>
```

```
Products</td>
```

```
{% endif %}
```

```
{% endif %}
```

```
</tr>
```

```
{% endfor %}
```

```
</table>
```

```


```

```
<input type="button" value="Home" onclick="window.location='/'">
```

```
<p></p>
```

```
{% endblock %}
```

SUPPLIERS					
ID	Name	Address		Telephone	<a href="#">Add supplier</a>
2	Supplier 1	Address 1a Address 1b Address 1c	<a href="#">Email</a>	1234	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>
3	Supplier 2	Address 2a Address 2b	<a href="#">Email</a>	2345	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>
4	Supplier 3	Address 3a Address 3b Address 3c	<a href="#">Email</a>	3456	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>
5	Supplier 4	Address 4a Address 4b	<a href="#">Email</a>	45	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>
6	Supplier 5	Address 5a Address 5b	<a href="#">Email</a>	567	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>
7	Supplier 6	Address 6a Address 6b Address 6c Address 6d	<a href="#">Email</a>	678	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>
8	Supplier 7	Address 7a	<a href="#">Email</a>	789	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Products</a>

Home

### 13.7.2 supplier\_add.html

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

{% block scripts %}
<script>
 function UpdateHidden()
 {
 //textarea does not have a name attribute
 address = document.getElementById('taAddress').value;
 document.getElementById('hiddenAddress').value = address;
 }
</script>
{% endblock %}

{% block content %}

 <h2>ADD SUPPLIER</h2>

 <form action="/supplier_add_post/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Supplier name</td>
 <td><input name="name"></td>
 </tr>
 <tr>
 <td>User name</td>
 <td><input name="username" onkeypress="return event.charCode != 32"></td>
 </tr>
 <tr>
 <td>Password</td>
 <td><input type="password" name="password"
 onkeypress="return event.charCode != 32"></td>
 </tr>
 <!-- <tr>
 <td>Address</td>
 <td><input name="address" size="40"></td>
 </tr> -->
 <tr>
 <td>Address</td>
 <td>
 <textarea rows="5" cols="60" id="taAddress"
 onchange="UpdateHidden();"></textarea>
 <input type="hidden" id="hiddenAddress" name="address"></hidden>
 </td>
 </tr>
 <tr>
 <td>Email address</td>
 <td><input name="email"></td>
 </tr>
 <tr>
 <td>Phone number</td>
 <td><input name="phone"></td>
 </tr>
 <tr></tr>
 <tr>
 <td colspan="2" align="center">
 <p></p>
 <input type="submit" value="Add">
 <input type="button" value="Cancel" onclick="window.location='/supplier_list/'">
 <p></p>
 </td>
 </tr>
 </table>

 </form>

{% endblock %}
```

### ADD SUPPLIER

Supplier name

User name

Password

Address

Email address

Phone number

### 13.7.3 supplier\_edit.html

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

{% block scripts %}
<script>
 function UpdateHidden()
 {
 //textarea does not have a name attribute
 address = document.getElementById('taAddress').value;
 document.getElementById('hiddenAddress').value = address;
 }
</script>
{% endblock %}

{% block content %}
```

### EDIT SUPPLIER

Supplier name

Address 

Address 1a  
 Address 1b  
 Address 1c

Email address

Phone number

```

 <h2>EDIT SUPPLIER</h2>

 <form action="/supplier_edit_post/{{supplier.supplier_id}}/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Supplier name</td>
 <td><input name="name" value="{{ supplier.supplier_name }}"></td>
 </tr>
 <!-- <tr>
 <td>Address</td>
 <td><input name="address" size="40"></td>
 </tr> -->
 <tr>
 <td>Address</td>
 <td>
 <textarea rows="5" cols="60" id="taAddress"
 onchange="UpdateHidden();">{{supplier.address }}</textarea>
 <input type="hidden" id="hiddenAddress" name="address"
 value="{{supplier.address }}"></hidden>
 </td>
 </tr>
 <tr>
 <td>Email address</td>
 <td><input name="email" value="{{supplier.user.email}}"></td>
 </tr>
 <tr>
 <td>Phone number</td>
 <td><input name="phone" value="{{supplier.phone_number}}"></td>
 </tr>
 <tr>
 <td colspan="2">
 <p></p>
 <input type="submit" value="Submit">
 <input type="button" value="Cancel" onclick="window.location='/supplier_list/'">
 <p></p>
 </td>
 </tr>
 </table>

 </form>

{% endblock %}

```

### 13.7.4 supplier\_delete.html

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

{% block content %}

<h2>DELETE CUSTOMER</h2>

<form action="/customer_delete_post/{{ customer.customer_id }}"
method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete {{ customer.customer_name }}?</p>

 <div align="center">
 <input type="submit" class="submit-btn" value=" Yes ">
 <input type="button" class="submit-btn" value=" No "
 onclick="window.location='/customer_list/'">
 <p></p>
 </div>
</form>
{% endblock %}
```

**DELETE SUPPLIER**  
Are you sure you want to delete S11?

### 13.7.5 vw\_suppliers.html

```
from django.shortcuts import render
from django.contrib import messages
from app_clothing.models import Supplier
from . import vw_home
from django.db import IntegrityError
from django.contrib.auth.models import Group, Permission, User

def supplier_list(request):
 supplier_set = []
 if request.user.is_superuser:
 supplier_set = Supplier.objects.order_by('supplier_name')
 if request.user.groups.filter(name="Suppliers").exists():
 supplier_set = Supplier.objects.filter(user__id=request.user.id)
 context = {'supplier_set': supplier_set, }
 return render (request, "supplier_list.html", context)

def supplier_add(request):
 if request.user.is_superuser:
 return render (request, "supplier_add.html")
 return vw_home.show_message(request, "Not permitted")

def supplier_add_post(request):
 if request.method == 'POST':
 #Get values from the template
 username = request.POST['username']
 password = request.POST['password']
 name = request.POST['name']
 address = request.POST['address']
 address = address.replace('
', '\n')
 email = request.POST['email']
 phone = request.POST['phone']

 #Create new user and add the user to the appropriate group
 user = User.objects.create_user(username=username, password=password, email=email)
 #perm_change_supplier = Permission.objects.filter(codename='change_supplier').first()
 #user.user_permissions.add(perm_change_supplier)
 group = Group.objects.get(name="Suppliers")
 group.user_set.add(user)

 #Create new supplier
 supplier = Supplier(supplier_name=name, user_id=user.id, address= address, phone_number =
phone)
 supplier.save()
 return supplier_list(request)

def supplier_edit(request, supplier_id):
 supplier = Supplier.objects.get(supplier_id = supplier_id)
 if request.user.is_superuser \
 or request.user.id == supplier.user.id: #Current user is a supplier
 return render (request, "supplier_edit.html", {'supplier': supplier})
 return vw_home.show_message(request, "Not permitted")
```



```

def supplier_edit_post(request, supplier_id):
 if request.method == 'POST':
 supplier = Supplier.objects.get(supplier_id=supplier_id)
 supplier.supplier_name = request.POST['name']
 supplier.address = request.POST['address']
 supplier.phone_number = request.POST['phone']
 supplier.save()
 supplier.user.email = request.POST['email']
 supplier.user.save()
 return supplier_list(request)

def supplier_delete(request, supplier_id):
 if request.user.is_superuser:
 supplier = Supplier.objects.get(supplier_id = supplier_id)
 return render (request, "supplier_delete.html", {'supplier': supplier})
 return vw_home.show_message(request, "Not permitted")

def supplier_delete_post(request, supplier_id):
 supplier = Supplier.objects.get(supplier_id = supplier_id)
 if request.method == 'POST':
 try:
 user_id = supplier.user.id
 supplier.delete()
 user = User.objects.get(id = user_id)
 user.delete()
 except:
 #return vw_home.show_message(request, e.__cause__)
 return vw_home.show_message(request, "Cannot delete supplier '" \
 + supplier.supplier_name + "'.\nA related product exists.")
 return supplier_list(request)

```

## 13.8 Products

### 13.8.1 product\_list.html

PRODUCTS (3)

**Suppliers:**  
☒ Supplier 1   ☒ Supplier 2   ☒ Supplier 3   ☐ Supplier 4  
☐ Supplier 5   ☐ Supplier 6   ☐ Supplier 7

**Order by:**  
☒ Product code   ☐ Supplier   ☐ Price

Code	Description	Price	Supplier	
<a href="#">P1</a>	Product 1	100.00	Supplier 1	<a href="#">Add product</a> <a href="#">Edit Delete</a>
<a href="#">P2</a>	Product 2	200.00	Supplier 2	<a href="#">Edit Delete</a>
<a href="#">P2b</a>	Product 2b	202.00	Supplier 2	<a href="#">Edit Delete</a>
<a href="#">S2P1</a>	Product 1	100.00	Supplier 2	<a href="#">Edit Delete</a>
<a href="#">S2P2</a>	Product 2	2022.00	Supplier 2	<a href="#">Edit Delete</a>
<a href="#">S2P3</a>	Product 3	300.00	Supplier 2	<a href="#">Edit Delete</a>

```

{% extends "base.html" %}

{% block content %}

<h2>PRODUCTS (3)</h2>

<form id="id_home" action="/product_list/" method="post" enctype="multipart/form-data">
 {% csrf_token %}

 <!-- Suppliers -->
 <div style="position: relative; text-align: left;
 background-color: darkgoldenrod; color: white; font-weight: bold;">
 Suppliers:
 {% for supplier in suppliers %}
 {% if forloop.counter0|divisibleby:4 %}

 {% endif %}
 <input type="checkbox" name="checked_suppliers" value="{{supplier.supplier_id}}"
 {% if supplier.supplier_id in selected_suppliers %}

```

```

 checked
 {% endif %}
 onclick="document.forms[0].submit();"
 {{ supplier.supplier_name }}
  
{% endfor %}
</div>

<!-- Order by -->
<div style="position: relative; margin-bottom: 20px; text-align: left;
background-color:darkgoldenrod; color: white; font-weight: bold;">
  Order by:
 
 <input type="radio" name="order_by" value="product_code"
onclick="document.forms[0].submit();" {% if order_by == "product_code" %} checked {% endif %}>Product
code 
 <input type="radio" name="order_by" value="supplier" onclick="document.forms[0].submit();" {%
if order_by == "supplier" %} checked {% endif %}>Supplier 
 <input type="radio" name="order_by" value="price" onclick="document.forms[0].submit();" {% if
order_by == "price" %} checked {% endif %}>Price 
</div>

<!-- Column headers -->
<table>
 <tr>
 <th width="70px" align="left">Code</th>
 <th width="120px" align="left">Description</th>
 <th width="60px" align="right">Price</th>
 <th width="20px"></th>
 <th width="150px" align="left">Supplier</th>
 <td>Add product</td>
 <td></td>
 </tr>
</table>

<!-- Data rows -->
<table>
 {% for product in products %}
 <tr>
 <td width="70px">
 {{ product.product_code }}
 </td>
 <td width="120px">{{ product.description }}</td>
 <td width="60px" align="right">{{ product.price |floatformat:2 }}</td>
 <td width="20px"></td>
 <td width="150px">{{ product.supplier.supplier_name }}</td>

 <td>
 Edit
 </td>
 <td>
 Delete
 </td>

 </tr>
 {% endfor %}
</table>

<input type="button" value="Home" onclick="window.location='/'">
<p></p>

</form>

{% endblock %}

```

### 13.8.2 product\_add.html

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

{% block content %}

<h2>ADD PRODUCT</h2>

<form action="/product_add_post/" method="post">
 {% csrf_token %}
 <table>
 <tr>
 <td>Product code</td>
 <td><input name="product_code" ></td>
 </tr>
 <tr>
 <td>Description</td>
 <td><input name="product_description"></td>
 </tr>
 <tr>
 <td>Price</td>
 <td><input type="number" min="0.00" step="any"
 name="product_price"></td>
 </tr>
 {% if supplier_list %}
 <tr>
 <td>Supplier</td>
 <td>
 <select name="supplier_id">
 {% for supplier in supplier_list %}
 {% if product.supplier_id == supplier.supplier_id %}
 <option value="{{ supplier.supplier_id }}"
 selected>
 {{ supplier.supplier_name }}
 </option>
 {% else %}
 <option value="{{ supplier.supplier_id }}">
 {{ supplier.supplier_name }}
 </option>
 {% endif %}
 {% endfor %}
 </select>
 </td>
 </tr>
 {% endif %}
 {% if supplier %}
 <td>Supplier</td>
 <td>{{supplier.supplier_name }} </td>
 {% endif %}
 </tr></tr>
 <tr>
 <td colspan="2" align="center">
 <p></p>
 <input type="submit" value="Add">
 <input type="button" value="Cancel" onclick="window.location='/product_list/'">
 <p></p>
 </td>
 </tr>
</table>

</form>

{% endblock %}
```

**ADD PRODUCT**

Product code

Description

Price

Supplier Supplier 1 ▾

**ADD PRODUCT**

Product code

Description

Price

Supplier Supplier 1 ▾

- Supplier 1
- Supplier 2
- Supplier 3
- Supplier 4
- Supplier 5**
- Supplier 6
- Supplier 7

### 13.8.3 product\_edit.html

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

{% block content %}

<h2>EDIT PRODUCT</h2>

<form action="/product_edit_post/{{product.product_id}}/"
method="post" enctype="multipart/form-data">
 {% csrf_token %}
 <table>
 <tr>
 <td>Product code</td>
 <td><input name="product_code"
value="{{product.product_code}}"></td>
 </tr>
 <tr>
 <td>Description</td>
 <td><input name="product_description"
value="{{product.description}}"></td>
 </tr>
 <tr>
 <td>Price</td>
 <td><input type="number", min="0.00" step="any"
name="product_price"
value="{{product.price |
floatformat:2}}"></td>
 </tr>
 {% if supplier_list %}
 <tr>
 <td>Supplier</td>
 <td>
 <select name="supplier_id"
value="{{product.supplier.supplier_id}}">
 {% for supplier in supplier_list %}
 {% if product.supplier_id == supplier.supplier_id %}
 <option value="{{ supplier.supplier_id }}" selected>
 {{ supplier.supplier_name }}
 </option>
 {% else %}
 <option value="{{ supplier.supplier_id }}">
 {{ supplier.supplier_name }}
 </option>
 {% endif %}
 {% endfor %}
 </select>
 </td>
 </tr>
 {% endif %}
 {% if supplier %}
 <tr>
 <td>Supplier</td>
 <td>{{supplier.supplier_name }} </td>
 </tr>
 <tr>
 <td>Description file</td>
 <td><input type="file" name="description_file"></td>
 </tr>
 <tr>
 <td></td>
 <td>{% if product.description_file %}

 {{ product.description_file_name_only }}
 {% endif %}
 </td>
 </tr>
 </table>
</form>
```

**EDIT PRODUCT**

Product code

Description

Price

Supplier

Description file  No file selected.

Image file  No file selected.

**EDIT PRODUCT**

Product code

Description

Price

Supplier

Description file  file selected.

Image file  file selected.

```

 <tr>
 <td>Image file</td>
 <td><input type="file" name="image_file"></td>
 </tr>
 {% if product.image_file %}
 <tr>
 <td colspan="2"><img src='{{ product.image_file.url }}' alt='No image'
 style="margin:0;padding:0; max-width:400px; max-height:400px; border: 1px dashed;">
 </td>
 </tr>
 {% endif %}

 <tr></tr>
 <tr>
 <td colspan="2" align="center">
 <p></p>
 <input type="submit" value="Submit">
 <input type="button" value="Cancel" onclick="window.location='/product_list/'">
 <p></p>
 </td>
 </tr>
 </table>

</form>

{% endblock %}

```

### 13.8.4 product\_delete.html

```

{% extends "base.html" %}

{% block content %}

<h2>DELETE PRODUCT</h2>

<form action="/product_delete_post/{{ product.product_id
}}/" method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete {{
product.product_code }}?</p>

 <div align="center">
 <input type="submit" class="submit-btn" value=" Yes ">
 <input type="button" class="submit-btn" value=" No "
onclick="window.location='/product_list/'">
 <p></p>
 </div>
</form>
{% endblock %}

```

## DELETE PRODUCT

Are you sure you want to delete P2?

### 13.8.5 vw\_products.py

```
from django.shortcuts import render
from django.contrib import messages
from django.db import IntegrityError
from app_clothing.models import Product, Supplier
from . import vw_home

#product_list 3 (Product list with header of suppliers that can be selected - selection based on session variable)
def product_list_3(request):
 #All suppliers
 suppliers = Supplier.objects.all()

 #Selected suppliers - check if the session variable is defined
 # We use a session variable to remember the last selected suppliers in case the user goes to
 another screen and then returns here
 try:
 selected_suppliers = request.session['selected_suppliers']
 except:
 selected_suppliers = [] #Empty suppliers list
 #selected_suppliers = suppliers.values_list('supplier_id', flat=True) #Alternative - all suppliers
 try: order_by = request.session['order_by'];
 except: order_by = 'product_code'

 #Products of selected suppliers
 products = Product.objects.filter(supplier__supplier_id__in= selected_suppliers)\
 .order_by(order_by, 'product_code')

 #Check if the user changed the filter fields - overwrite selected suppliers
 if request.method=="POST" \
 and 'product_list' in request.path: #Check if the request did not come from, e.g. edit_post
 selected_suppliers_s = request.POST.getlist('checked_suppliers')
 selected_suppliers = [int(id) for id in selected_suppliers_s] #Map list of str to list of int
 request.session['selected_suppliers'] = selected_suppliers
 order_by = request.POST.get('order_by')
 request.session['order_by'] = order_by
 products = Product.objects.filter(supplier__supplier_id__in=selected_suppliers)\
 .order_by(order_by, 'product_code')
 #else request came from elsewhere - keep the initial products list

 #Render the html for the filled-in template
 context = {'suppliers': suppliers, 'selected_suppliers': selected_suppliers, 'products': products,\
 'order_by' : order_by}
 return render (request, "product_list_3.html", context)

def product_list_3b(request, supplier_id=-1):
 #Update the session variable for selected suppliers
 if supplier_id == -1:
 request.session['selected_suppliers'] = []
 else:
 request.session['selected_suppliers'] = [supplier_id]
 #Proceed to the standard view to filter the products and display them in a template
 return product_list_3(request)

#product_list 4 (Product list with header of suppliers that can be selected - selection based on query params)
def product_list_4(request, selected_suppliers = ""):

 suppliers = Supplier.objects.all()
 if selected_suppliers != "":
 selected_suppliers = selected_suppliers.split(',')
 #return vw_home.show_message(request, selected_suppliers)
 products = Product.objects.filter(supplier__supplier_id__in = selected_suppliers).order_by("description")
 else:
 products = []
 context = {'suppliers': suppliers, 'selected_suppliers': selected_suppliers, 'products': products, }
 return render (request, "product_list_3.html", context)

from django.shortcuts import get_object_or_404

def product_details(request, product_code='P2'):
 product = get_object_or_404(Product, product_code=product_code)
 return render(request, 'product_details.html',{ 'product': product})

def product_add(request):
 if request.user.is_superuser:
 suppliers = Supplier.objects.all();
 return render(request, 'product_add.html', {"supplier_list": suppliers})

 #if request.user.id in suppliers.values_list('user_id', flat=True): #Current user is a supplier
 if request.user.groups.filter(name="Suppliers").exists(): #Current user is a supplier
 supplier = Supplier.objects.get(user__id = request.user.id)
 return render(request, 'product_add.html', {'supplier': supplier})

 return vw_home.show_message(request,"Not permitted")
```

```

def product_add_post(request):
 if request.method == 'POST':
 pCode = request.POST['product_code']
 description = request.POST['product_description']
 price = request.POST['product_price']
 if request.user.is_superuser:
 supplier_id = request.POST['supplier_id']
 else:
 supplier_id = Supplier.objects.get(user__id = request.user.id).supplier_id
 product = Product(product_code=pCode, description=description, price=price, supplier_id=supplier_id)
 product.save()
 return product_list_3(request)

def product_edit(request, product_id):
 product = Product.objects.get(product_id = product_id)

 if request.user.is_superuser:
 suppliers = Supplier.objects.all();
 return render(request, 'product_edit.html', {"product": product, "supplier_list": suppliers})

 if request.user.id == product.supplier.user.id: #Current user is the supplier of the product
 supplier = product.supplier
 return render(request, 'product_edit.html', {"product": product, "supplier": supplier})

 return vw_home.show_message(request, "Not permitted")

import os
from django.conf import settings
def product_edit_post(request, product_id):
 if request.method == 'POST':
 product = Product.objects.get(product_id=product_id)
 product.product_code = request.POST['product_code']
 product.description = request.POST['product_description']
 product.price = request.POST['product_price']
 if request.user.is_superuser:
 product.supplier_id = request.POST['supplier_id']
 else: pass #Supplier cannot change

 try:
 file= request.FILES['description_file']
 f_path1 = os.path.join(settings.MEDIA_ROOT, product.description_file.name)
 if os.path.isfile(f_path1): os.remove(f_path1)
 product.description_file = file
 except: pass

 try:
 file = request.FILES['image_file']
 f_path2 = os.path.join(settings.MEDIA_ROOT, product.image_file.name)
 if os.path.isfile(f_path2): os.remove(f_path2)
 product.image_file = file
 except: pass

 product.save()
 return product_list_3(request)

def product_delete(request, product_id):
 product = Product.objects.get(product_id = product_id)
 if request.user.is_superuser \
 or request.user.id == product.supplier.user.id: #Current user is the supplier of the product
 product = Product.objects.get(product_id = product_id)
 return render(request, 'product_delete.html', {"product": product})
 return vw_home.show_message(request, "Not permitted")

def product_delete_post(request, product_id):
 product = Product.objects.get(product_id = product_id)
 if request.method == 'POST':
 try:
 product.delete()
 except IntegrityError as e:
 messages.error(request, e.__cause__)
 return product_list_3(request)

```

## 13.9 Email

### 13.9.1 email.html

```
{% extends "base.html" %}
{% block content %}

<form action="/email_post/{{user.id}}/" method="post" enctype="multipart/form-data">
 {% csrf_token %}
 <h2 width="100%" align="center">Send email</h2>

 <table>
 <tr>
 <td>To</td>
 <td>{{user.username}}</td>
 </tr>
 <tr>
 <td>Subject</td>
 <td><input name="msg_subject"></td>
 </tr>
 <tr>
 <td>Message</td>
 <td><textArea name="msg_content" cols = "40" rows="10"></textArea></td>
 </tr>
 <tr>
 <td>Attach</td>
 <td><input type="file" name="attachment"></td>
 </tr>

 <tr>
 <td colspan="2" align="center">
 <input type="submit" value="Send">
 <input type="button" value="Cancel" onclick="window.location='/'>
 </td>
 </tr>
 </table>
</form>
{% endblock %}
```

### 13.9.2 vw\_email.py

```
from django.conf import settings
from django.shortcuts import render
#from django.core.mail import send_mail
from django.core.mail import EmailMessage
from . import vw_home, vw_customers
from django.contrib.auth.models import User
from app_clothing.forms import UploadFileForm

def email(request, user_id):
 user = User.objects.get(id = user_id)
 return render(request, 'email.html', {'user': user, })

def email_post(request, user_id):
 if request.method == 'POST':
 msg_from = settings.EMAIL_HOST_USER
 user = User.objects.get(id = user_id)
 msg_to = user.email
 msg_subject = request.POST['msg_subject']
 msg_content = request.POST['msg_content']
 recipient_list = [msg_to,]
 msg = EmailMessage(msg_subject, msg_content, msg_from, recipient_list)
 try:
 file = request.FILES['attachment']
 msg.attach(file.name, file.read()) #, file.content_type)
 except: pass
 msg.send()
 return vw_customers.customer_list(request)
```



## 13.10 Email

### 13.10.1 files.html

```
{% extends "base.html" %}
{% block scripts %}

<script>
 function OpenFile()
 {
 e = document.getElementById("file_list");
 var text = e.options[e.selectedIndex].text;
 //window.location = "\\file_open\\" + text + "\\"; //Same window
 window.open("\\file_open\\" + text + "\\"); //Other window
 } //OpenFile

 function DeleteFile()
 {
 e = document.getElementById("file_list");
 var text = e.options[e.selectedIndex].text;
 window.location = "\\file_delete\\" + text + "\\"; //Same window
 }
</script>

{% endblock %}
{% block content %}

<h2>FILES WAREHOUSE</h2>

<form action="/files_post/" method="post" enctype="multipart/form-data">
 {% csrf_token %}

 <!-- Download -->
 <h3 align="left">Available files</h3>
 <table>
 <tr>
 <td>
 <select id="file_list" value="uploaded_files">
 {% for file in uploaded_files %}
 <option>{{file}}</option>
 {% endfor %}
 </select>
 </td>
 </tr>
 <tr>
 <td>
 <input type="button" value="Open/Download" onclick="OpenFile();">
 <input type="button" value="Delete" onclick="DeleteFile();">
 </td>
 </tr>
 </table>

 <!-- Uploads -->
 <h3 align="left">Upload new file</h3>
 <table>
 <tr>
 <td>Previous upload:</td>
 <td>{%if uploaded_file %}
 {{uploaded_file}}
 {%endif%}
 </td>
 </tr>

 <tr>
 <td>Upload file:</td>
 <td><input type="file" name="upload_file" onchange="this.form.submit();"></td>
 </tr>
 </table>
 <p></p>
 <input type="button" value="Home" onclick="window.location='/'">
 <p></p>

</form>

{% endblock %}
```

### 13.10.2 file\_delete.html

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

{% block content %}

<h2>DELETE FILE</h2>

<form action="/file_delete_post/{{ file_name }}" method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete '{{ file_name }}'?</p>

 <div align="center">
 <input type="submit" class="submit-btn" value=" Yes ">
 <input type="button" class="submit-btn" value=" No "
 onclick="window.location='/files/'">
 <p></p>
 </div>
</form>
{% endblock %}
```

### 13.10.3 vw\_files.py

```
from django.conf import settings
from django.core.files.storage import FileSystemStorage
from django.http import FileResponse
from django.shortcuts import render

import os
from os import listdir
from os.path import isfile, join

def uploaded_files():
 media_path = settings.MEDIA_ROOT
 list = listdir(media_path)
 files = []
 for e in list:
 if isfile(join(media_path, e)):
 files.append(e)
 return files

def files(request):
 return render(request, 'files.html', {'uploaded_files': uploaded_files()})

def files_post(request):
 if request.method == 'POST':
 try:
 file=request.FILES['upload_file']
 f_path = os.path.join(settings.MEDIA_ROOT, file.name)
 if os.path.isfile(f_path): os.remove(f_path)
 fs = FileSystemStorage()
 fs.save(file.name, file)
 return render(request, 'files.html', {'uploaded_file': file.name, 'uploaded_files':
uploaded_files()})
 except: pass
 return files(request)

def file_open(request, file_name):
 f_path = os.path.join(settings.MEDIA_ROOT, file_name)
 if os.path.isfile(f_path):
 content = open(f_path, 'rb')
 response = FileResponse(content)
 return response
 return files(request)

def file_delete(request, file_name):
 return render(request, "file_delete.html", {'file_name': file_name})

def file_delete_post(request, file_name):
 if request.method=="POST":
 f_path = os.path.join(settings.MEDIA_ROOT, file_name)
 if os.path.isfile(f_path):
 os.remove(f_path)
 return files(request)
```

## 13.11 Serialization

### 13.11.1 serializers\_book.py

```
from rest_framework.views import APIView, status, Response
from rest_framework import serializers

from app_clothing.models import Book

class BookSerializer(serializers.ModelSerializer):
 class Meta:
 model = Book
 fields = ["book_id", "title", "author"]

class BookList(APIView):

 def get(self, request, *args, **kwargs):
 books = Book.objects.all()
 serializers = BookSerializer(books, many=True)
 return Response({"status": "success", "books": serializers.data}, status=200)

 def post(self, request, *args, **kwargs):
 #Request data from template and build data object
 data = {
 "title" : request.data.get("title"),
 "author" : request.data.get("author"),
 }

 #Serialize and save
 serializer = BookSerializer(data=data)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_201_CREATED)
 return Response(serializer.errors, status =status.HTTP_400_BAD_REQUEST)

class BookDetails(APIView):

 def get(self, request, book_id, *args, **kwargs):
 try:
 book = Book.objects.get(book_id=book_id)
 serializer= BookSerializer(book) #Note the difference here
 return Response({"status": "success", "book": serializer.data}, status=200)
 except:
 return Response({"res": "Book with the given id does not exist"}, \
 status=status.HTTP_400_BAD_REQUEST)

 def put(self, request, book_id, *args, **kwargs):
 #return show_message(request, book_id)
 try:
 book = Book.objects.get(book_id=book_id)

 data = {
 "title": request.data.get("title"),
 "author" : request.data.get("author")
 }

 serializer = BookSerializer(instance=book, data=data, partial=True)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_200_OK)
 return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
 except:
 return Response({"res": "Book with given id does not exist"}, \
 status=status.HTTP_400_BAD_REQUEST)

 def delete(self, request, book_id, *args, **kwargs):
 try:
 book = Book.objects.get(book_id=book_id)
 book.delete()
 return Response({"res": "Object deleted!"}, status=200)
 except:
 return Response({"res": "Book with given id does not exist"}, \
 status=status.HTTP_400_BAD_REQUEST)
```

### 13.11.2 serializers\_user.py

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

class UserSerializer(serializers.ModelSerializer):
 class Meta:
 model = User
 fields = ["id", "username", "email"]
```

### 13.11.3 serializers\_supplier.py

```
from app_clothing.views.vw_home import show_message
from app_clothing.models import Supplier
from django.contrib.auth.models import User

from rest_framework.views import APIView, status, Response
from rest_framework import serializers

from app_clothing.views.serializers_user import UserSerializer

class SupplierGetSerializer(serializers.ModelSerializer):
 user = UserSerializer()

 class Meta:
 model = Supplier
 fields = ["supplier_id", "user", "supplier_name", "address", "phone_number"]

class SupplierPutSerializer(serializers.ModelSerializer):

 class Meta:
 model = Supplier
 fields = ["supplier_id", "user", "supplier_name", "address", "phone_number"]

class SupplierList(APIView):

 def get(self, request, *args, **kwargs):
 result = Supplier.objects.all()
 serializers = SupplierGetSerializer(result, many=True)
 return Response({"status": "success", "suppliers": serializers.data}, status=200)

 def post(self, request, *args, **kwargs):
 #Request data from template and build data object
 data = {
 "user": 1,
 "supplier_name" : request.data.get("supplier_name"),
 "address" : request.data.get("address"),
 "phone_number" : request.data.get("phone_number"),
 }

 #Serialize and save
 serializer = SupplierPutSerializer(data=data)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_201_CREATED)
 return Response(serializer.errors, status =status.HTTP_400_BAD_REQUEST)
```

```

class SupplierDetails(APIView):

 def get(self, request, supplier_id, *args, **kwargs):
 try:
 supplier = Supplier.objects.get(supplier_id=supplier_id)
 serializer= SupplierGetSerializer(supplier) #Note the difference here
 return Response({"status": "success", "supplier": serializer.data},
 status=200)
 except:
 return Response({"res": "Supplier with given id does not exist"},
 status=status.HTTP_400_BAD_REQUEST)

 def put(self, request, supplier_id, *args, **kwargs):

 try:
 supplier = Supplier.objects.get(supplier_id=supplier_id)
 data = {
 "user": request.data.get("user"),
 "supplier_name": request.data.get("supplier_name"),
 "address" : request.data.get("address"),
 "phone_number": request.data.get("phone_number")
 }

 serializer = SupplierPutSerializer(instance=supplier, data=data, partial=True)
 if serializer.is_valid():
 serializer.save()
 return Response(serializer.data, status=status.HTTP_200_OK)
 return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
 except:
 return Response({"res": "Supplier with given id does not exist"},
 status=status.HTTP_400_BAD_REQUEST)

 def delete(self, request, supplier_id, *args, **kwargs):
 try:
 supplier = Supplier.objects.get(supplier_id=supplier_id)
 supplier.delete()
 return Response({"res": "Object deleted!"}, status=200)
 except:
 return Response({"res": "Supplier with given id does not exist"},
 status=status.HTTP_400_BAD_REQUEST)

```