

# Python for Beginners

Web Programming with Django

Basic Concepts

# Web Programming with Python

You can use several web frameworks with python:

- Django

- very popular web framework
- All included (e.g. ORM)
- Ready to use admin pages
- install the latest version of Django with PIP:

```
python -m pip install django
```

- For more details see:

<https://www.djangoproject.com>

- Flask

- younger web framework, fast growing community
- Very flexible and modular
- Ideal for experts knowing what they need and want to do
- Install the latest version of Flask with PIP:

```
python -m pip install flask
```

- For more details see:

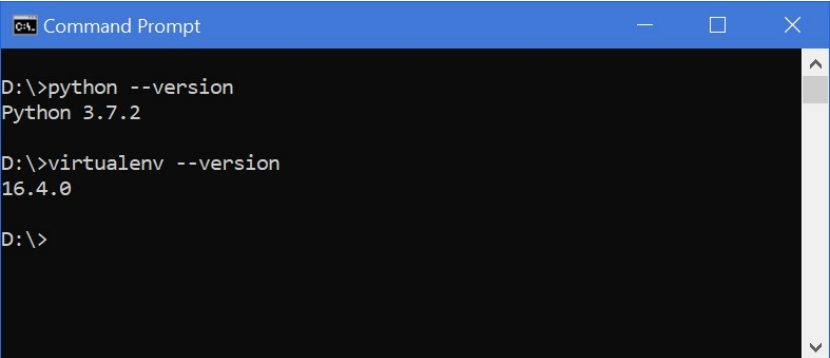
<http://flask.pocoo.org>

# Creating a new Django Project

- Before you start:

## Check your Python installation

- Can you run the python command and virtualenv command from any terminal or command prompt?
- If not: set the environment variable (Umgebungsvariable) PATH:
- Open a command prompt and type in: `SystemPropertiesAdvanced.exe`
- Add two entries to the PATH Variable:
  - The path to the directory of python.exe
  - The path to the directory Scripts inside the directory of python.exe



```
C:\> Command Prompt

D:\>python --version
Python 3.7.2

D:\>virtualenv --version
16.4.0

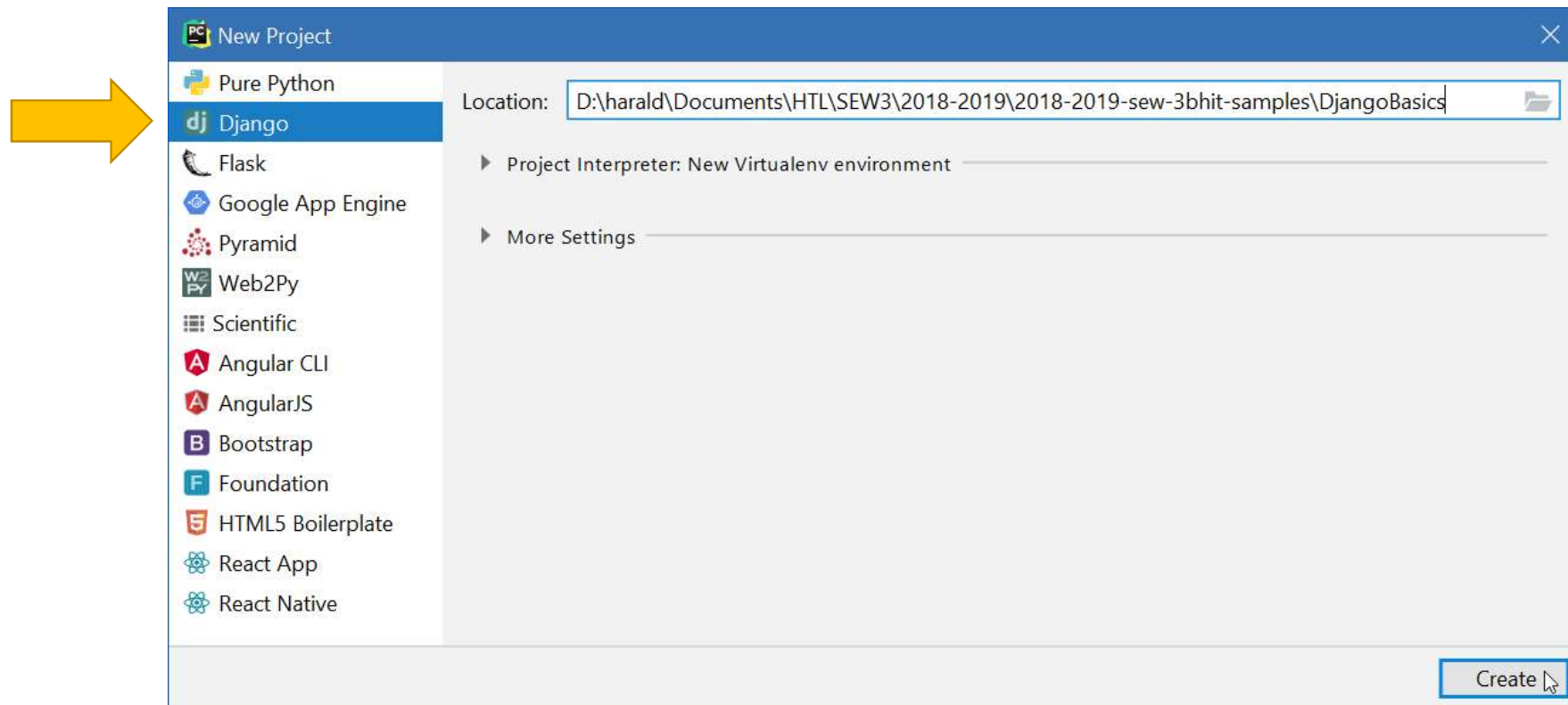
D:\>
```

You should find you python or anaconda installation in

`C:\Users\<your username>\AppData\Local\Programs\Python\Python37)`

# Creating a new Django Project with PyCharm

- Create a new Django Project:

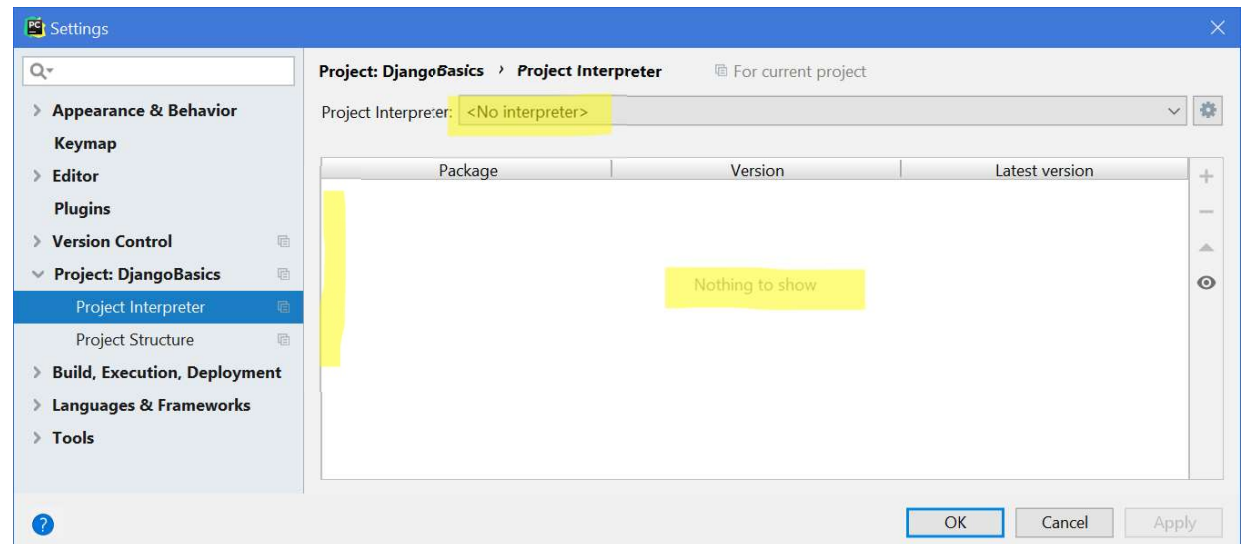


# Fixing the Project Configuration and Virtual Environment

If you use an existing project e.g. checked out with git:

- **Step 1:** Clean up old settings if there are some:

- Be sure that no old/wrong virtual environment settings are active. Therefore open your terminal and check if you see a “venv” in front of the prompt. If this is the case open File | Settings and set the project interpreter to <No interpreter>:



→ Now your terminal should start without virtual environment

# Fixing the Project Configuration and Virtual Environment

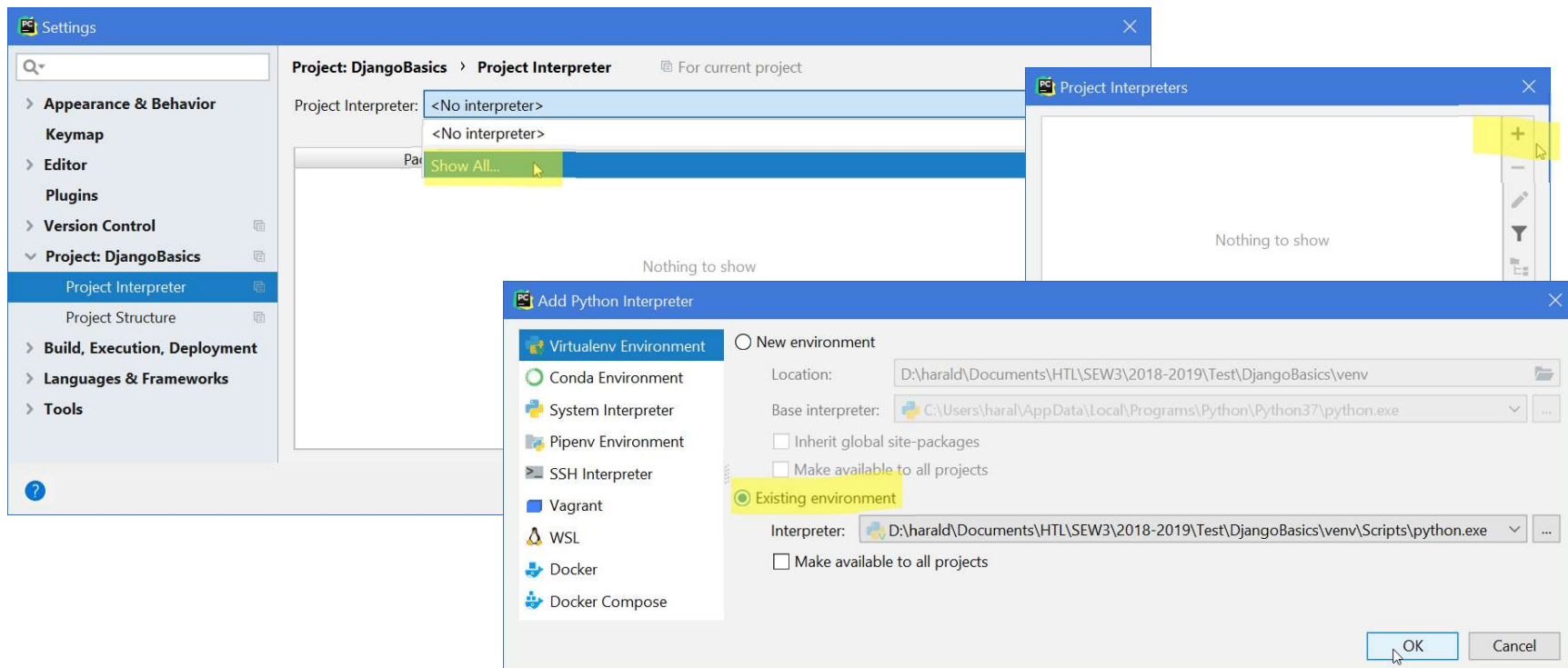
- **Step 2:** Recreate the virtual environment “virtualenv”
  1. Open a terminal
  2. Call the script `virtualenv env` → this creates the `venv` folder

```
D:\harald\Documents\HTL\SEW3\2018-2019\Test\DjangoBasics>virtualenv venv
Using base prefix 'c:\\users\\harald\\appdata\\local\\programs\\python\\python37'
New python executable in D:\harald\Documents\HTL\SEW3\2018-2019\Test\DjangoBasics\venv\Scripts\python.exe
Installing setuptools, pip, wheel...
done.

D:\harald\Documents\HTL\SEW3\2018-2019\Test\DjangoBasics>
```

# Fixing the Project Configuration and Virtual Environment

- **Step 3:** Select the python interpreter from your virtual environment:



# Fixing the Project Configuration and Virtual Environment

- Close and open your terminal
  - your terminal should use the virtual environment created in Step 2
- Now you can install all your needed libraries
  - manually by calling `pip install libraryname`
  - or if you have a list of libraries in `requirements.txt` you can install them by calling `pip install -r requirements.txt`

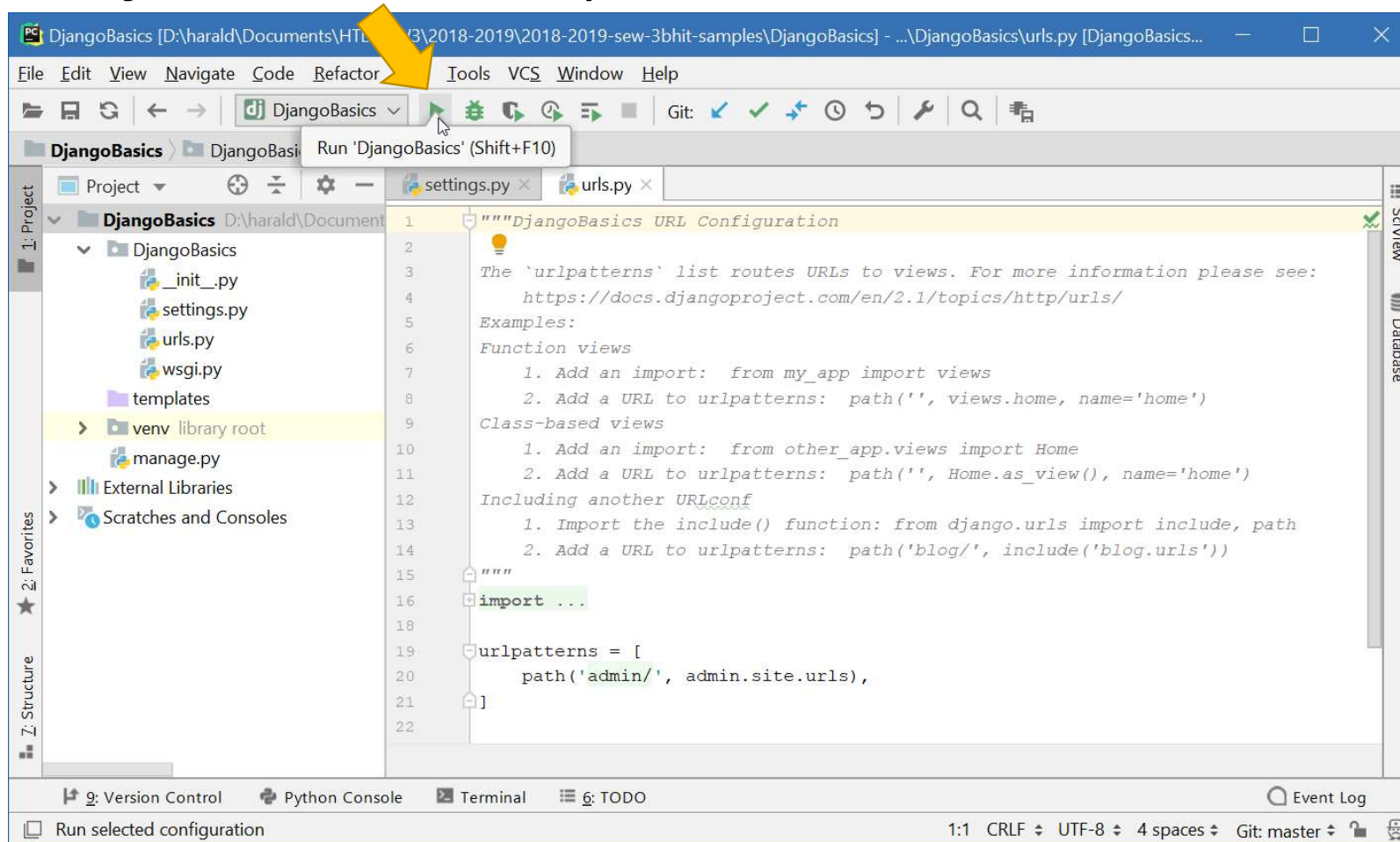
```
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. All rights reserved.

(venv) D:\harald\Documents\HTL\SEW3\2018-2019\Test\DjangoBasics>pip install -r requirements.txt
Collecting Django==2.1.7 (from -r requirements.txt (line 1))
  Using cached https://files.pythonhosted.org/packages/c7/87/fbd666c4f87591ae25b7bb374298e8629816e87193c4099d3608ef11fab9/Django-2.1.7-py3-none-any.whl
Collecting Faker==1.0.2 (from -r requirements.txt (line 2))
  Using cached https://files.pythonhosted.org/packages/79/36/8elaa2f775018ea11a897bef32b6f80d78dcb6cc6563744f2e1dcf128b82/Faker-1.0.2-py2.py3-none-any.whl
Collecting python-dateutil==2.8.0 (from -r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/41/17/c62facbfbfd163c7f57f3844689e3a78bae1f403648a6afb1d0866d87fbb/python_dateutil-2.8.0-py2.py3-none-any.whl
Collecting pytz==2018.9 (from -r requirements.txt (line 4))
  Using cached https://files.pythonhosted.org/packages/61/28/1d3920e4d1d50b19bc5d24398a7cd85cc7b9a75a490570d5a30c57622d34/pytz-2018.9-py2.py3-none-any.whl
Collecting six==1.12.0 (from -r requirements.txt (line 5))
  Using cached https://files.pythonhosted.org/packages/73/fb/00a976f728d0d1fecfe898238ce23f502a721c0ac0ecfedb80e0d88c64e9/six-1.12.0-py2.py3-none-any.whl
Collecting text-unidecode==1.2 (from -r requirements.txt (line 6))
  Using cached https://files.pythonhosted.org/packages/79/42/d717cc2b4520fb09e45b344b1b0b4e81aa672001dd128c180fab655c341/text_unidecode-1.2-py2.py3-none-any.whl
Installing collected packages: pytz, Django, six, python-dateutil, text-unidecode, Faker
Successfully installed Django-2.1.7 Faker-1.0.2 python-dateutil-2.8.0 pytz-2018.9 six-1.12.0 text-unidecode-1.2

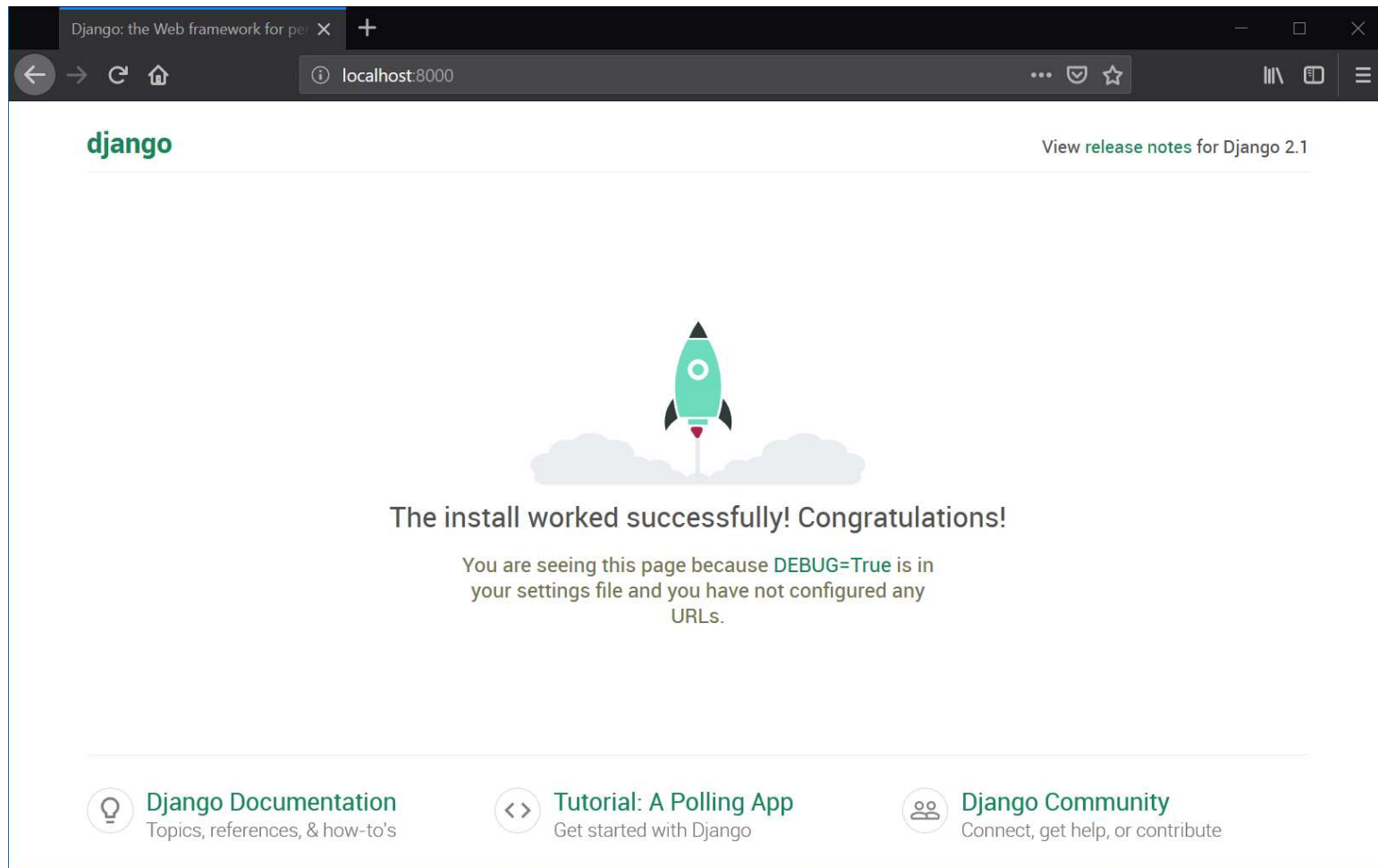
(venv) D:\harald\Documents\HTL\SEW3\2018-2019\Test\DjangoBasics>
```



# Project Directory Structure and Run Server



# Default Django Landing Page



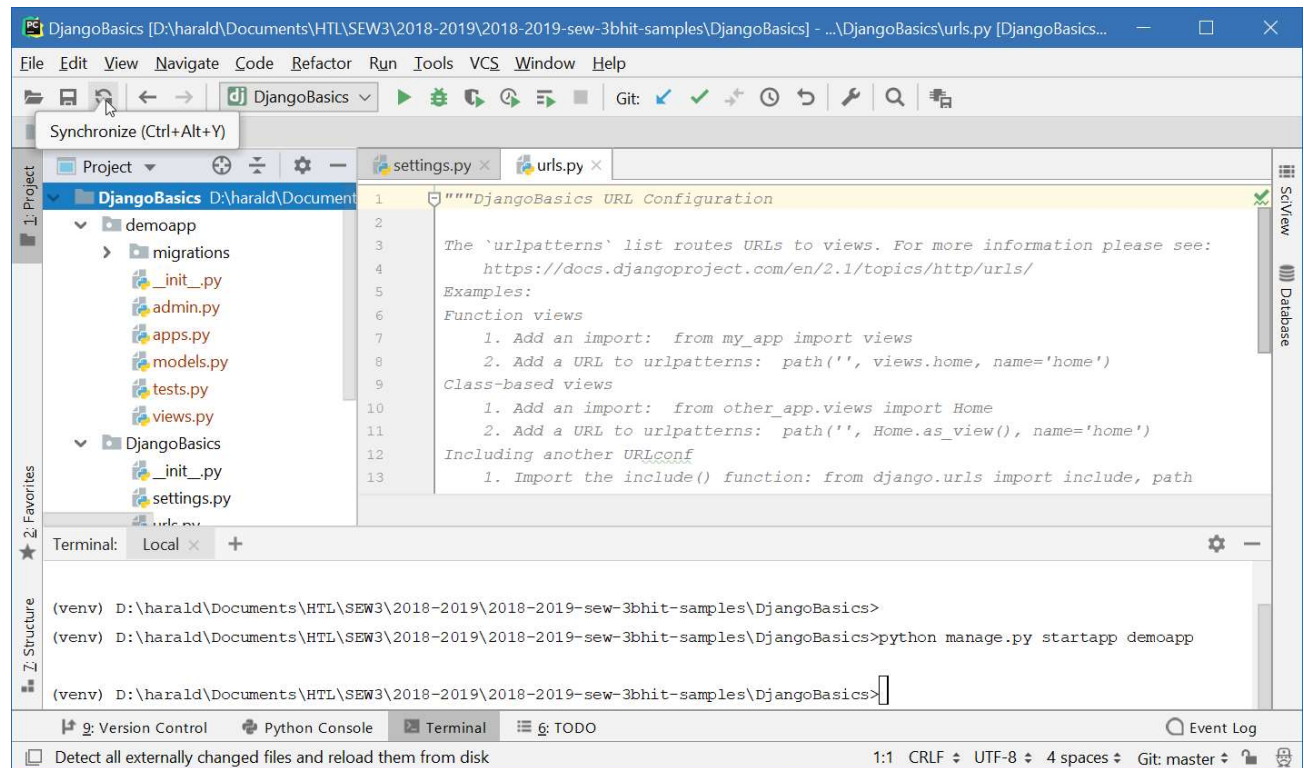
# Create a Django Application

- In PyCharm open the terminal and type:

```
python manage.py startapp demoapp
```

where demoapp is the name of your app

- A Django App is a kind of module with a specific purpose
- Press the synchronize Button to refresh the project tree with the created app



# Add the application

- Open the settings.py file and add the demoapp to the INSTALLED\_APPS

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

# Structure of a Django App

File \ Folder	Role
apps.py	Configuration and initialization
models.py	Data layer
admin.py	Administrative interface
urls.py	URL routing
views.py	Control Layer
tests.py	Test the app
migrations\	Holds migration files

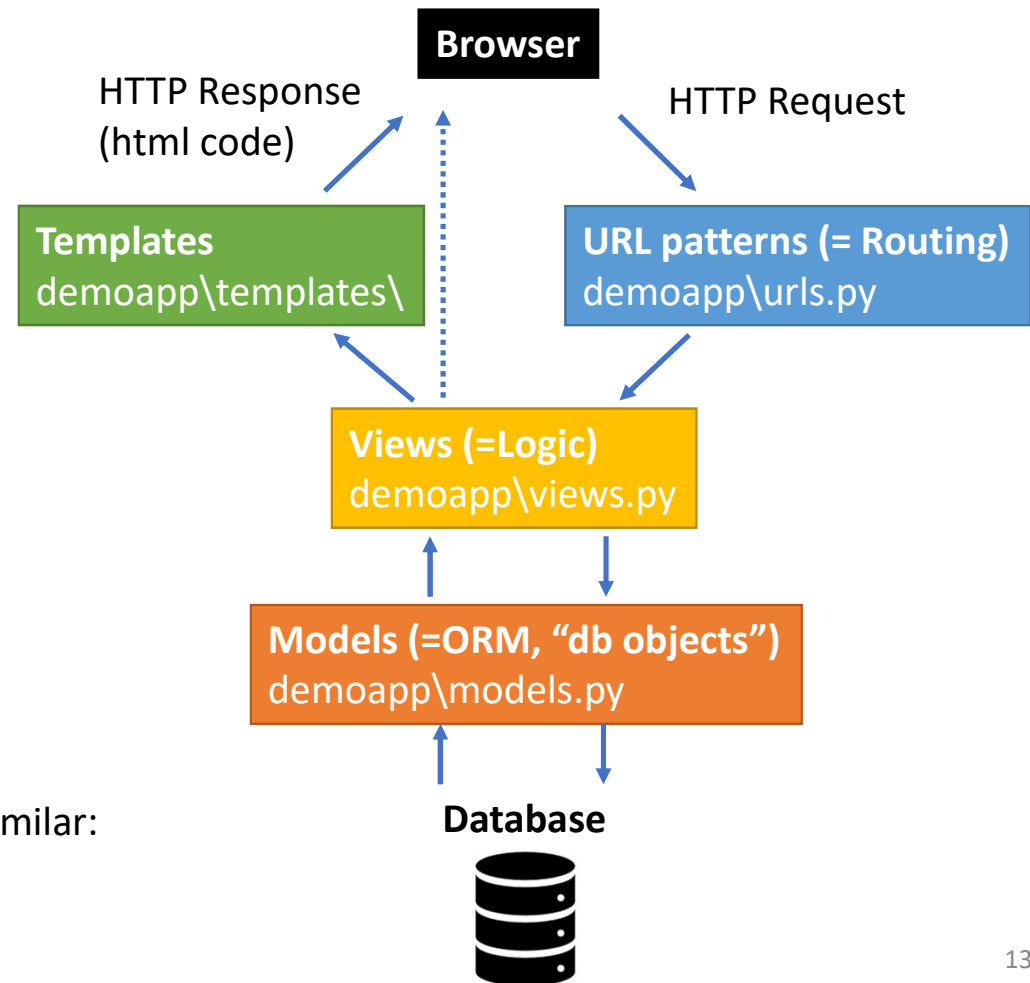
The Django Structure follows the Model View Template pattern (MVT)

The Model View Controller pattern (MVC) is quite similar:

M = M

V = C

T = V



# Models

- Are part of the data layer of an app (ORM)
- Define database structure (db schema)
- Allow us to query the database (query builder)
- The `models.py` file contains the set of models for the Django app.
- A model class is a class inheriting from `django.db.models.Model` and is used to define fields as class attributes.

# Model Classes

- In `models.py` create a Model Class 'Person' with some fields

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

```
class Person(models.Model):  
    SEX_CHOICES = [('M', 'Male'), ('F', 'Female')]  
    firstname = models.CharField(max_length=100)  
    lastname = models.CharField(max_length=100)  
    age = models.IntegerField(null=True)  
    sex = models.CharField(choices=SEX_CHOICES, max_length=1)  
    description = models.TextField(blank=True)  
  
    def __str__(self):  
        return self.firstname + ' ' + self.lastname
```

# Migrations

- Create the initial database structure (= db schema)
- Change (upgrade) the database structure to another version
  - Add a new model → add a new db table + relations
  - Add/change/remove a field → alter the db table + relations
- Every migration is a new version of the database structure

Perform the two yellow commands:

`python manage.py makemigrations` ... generates the migration scripts from the models

`python manage.py migrate` ... performs all migrations to the database

Some further (optional) commands:

`python manage.py showmigrations` ... shows the status of the migrations

`python manage.py sqlmigrate demoapp 0001` ... shows the sql statements of the migration 0001

`python manage.py migrate demoapp zero` ... resets the migrations of the demoapp



# Admin Interface

The admin interface is an easy way to manage your models

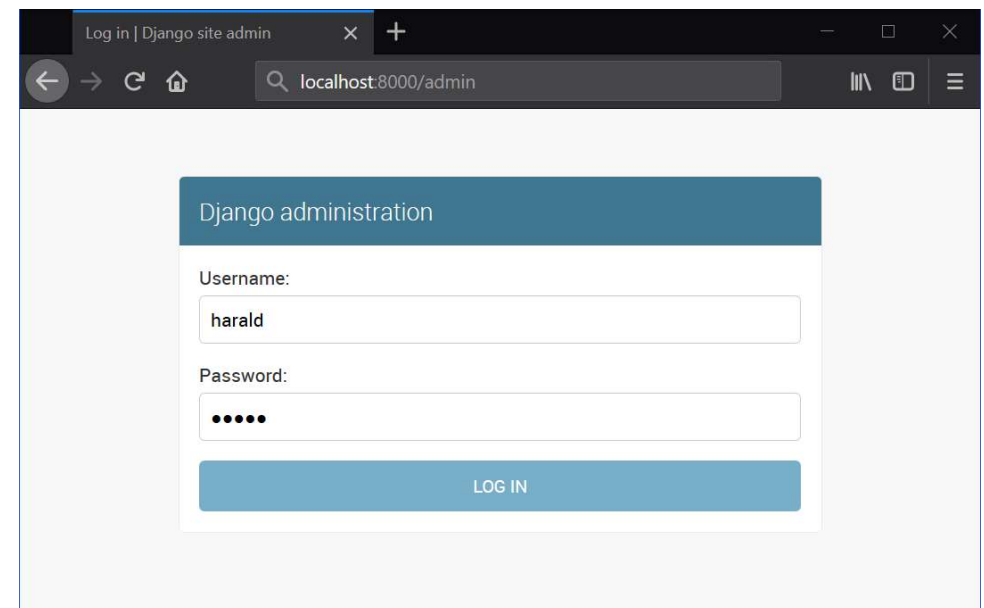
- Open the file `admin.py` in your app and edit

```
from django.contrib import admin
from .models import Person
```

```
admin.site.register(Person)
```

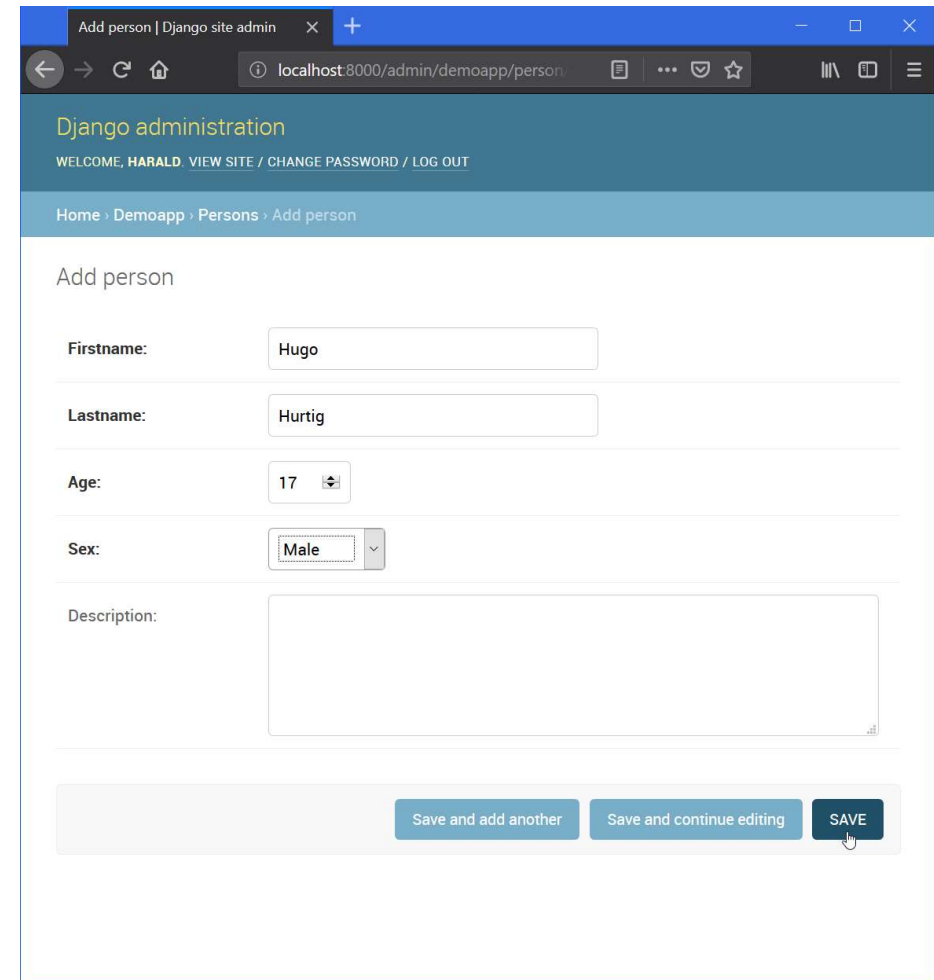
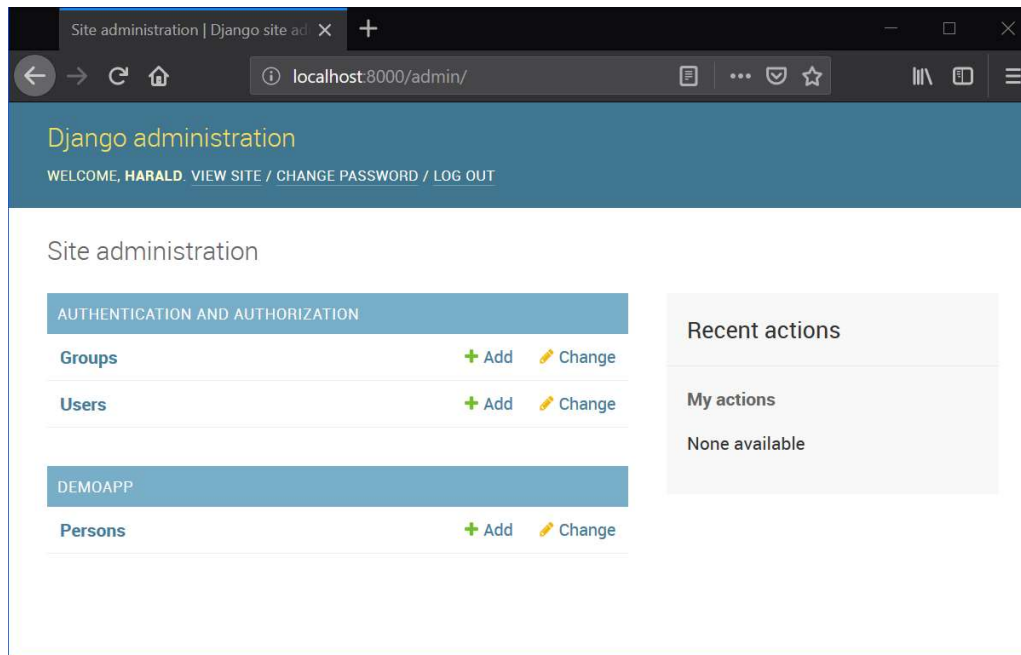
- In the terminal create a superuser that can login to the admin interface:

```
python manage.py createsuperuser
```



# Using the Admin Interface

Here you can show, edit, create and delete Person objects



# URL Patterns and Routing

To access the web app you have to define routes to view functions that returns template pages as response:

- `http://localhost:8000/people/`

`people/` → `views.index` → `people\index.html`

- `http://localhost:8000/people/1/`

`people/<int:person_id>/` → `views.person_detail` → `people\person_detail.html`



request: matches to URL pattern



called views function



response: processed template page

# URL Patterns and Routing

- Import the views and route to them

```
from django.contrib import admin
from django.urls import path
```

```
from demoapp import views
```

```
urlpatterns = [
    # the predefined routes to admin urls
    path('admin/', admin.site.urls),

    # 1.) people path routes to index
    path('people/', views.index, name='index_route'),

    # 2.) people path followed by the id routes to the person details with this id
    path('people/<int:person_id>', views.person_detail, name='person_detail_route'),
]
```

# Views

- Contains functions that process the routed requests
- The functions implement the logic of the web app and return the response to the browser (= what you will view/see)
- The functions should not build the response by themselves but use templates that help building the response

# Views

Edit the file demoapps\views.py:

```
from django.shortcuts import render
from django.http import Http404

from .models import Person

# the index view returns all people with the index template
def index(request):
    people = Person.objects.all() # get all Person objects from the database
    # return the rendered template page with the passed parameter people
    return render(request, 'person/index.html', {'people': people})

# the person_detail view returns one person with the detail template
def person_detail(request, person_id):
    try:
        # try to get the Person object with the person_id
        person = Person.objects.get(id=person_id)
    except Person.DoesNotExist:
        # if the Person object was not found raise a 404 Http error
        raise Http404('Person not found')
    # return the rendered template page with the passed parameter person
    return render(request, 'person/detail.html', {'person': person})
```

# Templates

- A template is a special html page
  - with variables `{{ ... }}` (=placeholders)
  - tags `{% ... %}` (= control structures, blocks for reuse etc.)
  - and comments `{# ... #}`
- The template is rendered before it is sent back to the browser (=response)
- Django Template Language (DTL):  
<https://docs.djangoproject.com/en/2.1/ref/templates/language/>

# Index Template



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Alle Personen</title>
</head>
<body>
  <h1>All People</h1>
  <ul>
    {% for person in people %} {# repeat for every person in people #}
    <li>
      <a href="{% url 'person_detail_route' person.id %}">
        {{ person.firstname }} {{ person.lastname }}
      </a>
    </li>
    {% endfor %}
  </ul>
  Number of People: {{ people.count }}
</body>
</html>
```



# Detail Template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Person Details</title>
</head>
<body>
<h1>Person Details:</h1>
  <ul>
    <li>Id: {{ person.id }}</li>
    <li>Firstname: {{ person.firstname }}</li>
    <li>Lastname: {{ person.lastname }}</li>
    <li>Age: {{ person.age }}</li>
    <li>Sex: {{ person.get_sex_display }}</li>
    {% if person.description %} {# optional #}
      <li>Description: {{ person.description }}</li>
    {% endif %}
  </ul>
  <a href="{{ request.META.HTTP_REFERER }}">Go back</a>
</body>
</html>
```

# Creating and Manipulating the Models

- Apart from the admin interface many web apps should be able to
  - Create new model objects
  - Modify existing model objects
  - Delete existing model objects
- Therefore further views functions with template pages must be implemented:
  -  • person\_create ... returns an empty html form to enter the new person data
  - person\_store ... creates a new person object and stores it to the database
  -  • person\_edit ... returns a html form filled with the person data to change
  - person\_update ... updates the data of the person object in the database
  - person\_destroy ... removes the person object from the database

# Create a new Person

- First create a new views function in demoapp\views.py:

```
# the person_create view returns an empty form to enter the new person data  
def person_create(request):  
    return render(request, 'person/create.html')
```

- Add a route to the function in the urls.py:

```
# people followed by create routes to the person create page  
path('people/create/', views.person_create, name='person_create_route'),
```

- And insert a link to the person\_create\_route in the index.html template page

```
<p>  
    <a href="{% url 'person_create_route' %}">Create</a>  
</p>
```

- Next: create the html form for entering the new person data

# The Create Person Form

Every html form has

- a method attribute (=GET or POST)  
→ Default is GET, but always use POST if you change something
- an action attribute (=url to submit)  
→ Do not hard code the URL.  
Always use a DTL tag with the route name
- several input tags  
[https://www.w3schools.com/html/html\\_form\\_input\\_types.asp](https://www.w3schools.com/html/html_form_input_types.asp)  
→ Every input must have a name  
otherwise no data will be submitted
- and don't forget the **csrf token**!  
→ Cross-site request forgery (CSRF)  
Secured via a hidden input field carrying a unique number for every submit request  
<https://docs.djangoproject.com/en/2.1/ref/csrf/>

```
<h2>Create a new Song</h2>
<form method="post" action="{% url 'person_store_route' %}">
  {% csrf_token %}
  <p>
    <strong>Firstname:</strong>
    <input type="text" name="firstname"/>
  </p>
  <p>
    <strong>Lastname:</strong>
    <input type="text" name="lastname"/>
  </p>
  <p>
    <strong>Age:</strong>
    <input type="text" name="age"/>
  </p>
  <p>
    <strong>Sex:</strong>
    <select name="sex">
      <option value="M">Male</option>
      <option value="F">Female</option>
    </select>
  </p>
  <p>
    <strong>Description:</strong>
    <textarea name="description" rows="5"></textarea>
  </p>
  <p>
    <button type="submit">Create</button>
  </p>
</form>
```

# Store the Person Data

- Add a route to the store function in urls.py:

```
# people followed by store routes to the person store function  
path('people/store/', views.person_store, name='person_store_route'),
```

- Implement the store function in demoapp\views.py:

```
# the person_store view stores the new person and redirects back to the person_index_route  
def person_store(request):  
    try:  
        p = Person()  
        p.firstname = request.POST.get('firstname')  
        p.lastname = request.POST.get('lastname')  
        p.age = request.POST.get('age')  
        p.sex = request.POST.get('sex')  
        p.description = request.POST.get('description')  
        p.save()  
        return redirect('person_index_route')  
    except Exception as ex:  
        return HttpResponseRedirect(ex)
```

# Edit and Update an Existing Person

- Add the route to urls.py:

```
# people followed by edit and the id routes to edit form
path('people/edit/<int:person_id>/', views.person_edit, name='person_edit_route'),
```

- Insert this link to every person in the index.html template page:

```
<a href="{% url 'person_edit_route' person.id %}">Edit</a>
```

- Create a views function in demoapp\views.py:

```
def person_edit(request, person_id):
    try:
        # try to get the Person object with the person_id
        person = Person.objects.get(id=person_id)
    except Person.DoesNotExist:
        # if the Person object was not found raise a 404 Http error
        raise Http404('Person not found')
    return render(request, 'person/edit.html', {'person': person})
```

# The Edit Person Form

- The form is similar to the create form but fills all inputs with the current field values of the person object:

```
<h1>Edit Person Details:</h1>
<form method="post" action="{% url 'person_update_route' person.id %}">
  {% csrf_token %}
  <p>Id: {{ person.id }}</p>
  <p>Firstname: <input name="firstname" value="{{ person.firstname }}" /></p>
  <p>Lastname: <input name="lastname" value="{{ person.lastname }}" /></p>
  <p>Age: <input name="age" value="{{ person.age }}" /></p>
  <p>Sex:
    <select name="sex">
      {% for key,value in person.SEX_CHOICES %}
        <option value="{{ key }}" {% if person.sex == key %} selected {% endif %}>
          {{ value }}
        </option>
      {% endfor %}
    </select>
  </p>
  <p>Description: <textarea name="description" rows="5">{{ person.description }}</textarea></p>
  <button type="submit">Save</button>
  &nbsp;
  <a href="{% url 'person_index_route' %}">All People</a>
</form>
```

# Update the Modified Person Data

- Add a route to the update function in urls.py:

```
# people followed by update and the id routes to the update function
path('people/update/<int:person_id>/', views.person_update, name='person_update_route'),
```

- Implement the update function in demoapp\views.py:

```
def person_update(request, person_id):
    try:
        p = Person.objects.get(id=person_id)
        p.firstname = request.POST.get('firstname')
        p.lastname = request.POST.get('lastname')
        p.age = request.POST.get('age')
        p.sex = request.POST.get('sex')
        p.description = request.POST.get('description')
        p.save()
    except Person.DoesNotExist:
        raise Http404('Person not found')
    except Exception as ex:
        return HttpResponseBadRequest(ex)
    return redirect('person_index_route')
```



# Delete a Person

- Add the route to urls.py:

```
#people followed by delete and the id routes to the delete function  
path('people/delete/<int:person_id>/', views.person_delete, name='person_delete_route'),
```

- Insert this mini-form to every person in the index.html template page:

```
<form method="post" action="{% url 'person_delete_route' person.id %}">  
    {% csrf_token %}  
    <button type="submit">Delete</button>  
</form>
```

# Delete the Selected Person

- Add this function to demoapp\views.py:

```
def person_delete(request, person_id):  
    try:  
        # try to get the Person object with the person_id  
        person = Person.objects.get(id=person_id)  
        person.delete()  
    except Person.DoesNotExist:  
        # if the Person object was not found raise a 404 Http error  
        raise Http404('Person not found')  
    except Exception as ex:  
        return HttpResponseRedirect(ex)  
    return redirect('person_index_route')
```

# Show all People as Table

```
<h1>All People</h1>
<table border="1" cellpadding="5">
  <thead>
    <td>Firstname</td>
    <td>Lastname</td>
    <td colspan="3">Actions</td>
  </thead>
  {% for person in people %} {# repeat for every person in people #}
  <tr>
    <td>{{ person.firstname }}</td>
    <td>{{ person.lastname }}</td>
    <td><a href="{% url 'person_detail_route' person.id %}">Show Details</a></td>
    <td><a href="{% url 'person_edit_route' person.id %}">Edit</a></td>
    <td>
      <form method="post" action="{% url 'person_delete_route' person.id %}">
        {% csrf_token %}
        <button type="submit">Delete</button>
      </form>
    </td>
  </tr>
  {% endfor %}
</table>
```

# Brush up your App with Twitter Bootstrap 4

- Bootstrap is of the standard frameworks for fast and easy web development
- includes HTML and CSS based design templates for typography, forms, buttons, tables, navigation, modals, image carousels and many other, as well as optional JavaScript plugins
- Bootstrap also gives you the ability to easily create responsive designs
- Use these tutorials/links to learn bootstrap:  
<https://www.w3schools.com/bootstrap4>  
<https://getbootstrap.com/>  
<http://holdirbootstrap.de/css/>  
<https://medium.freecodecamp.org/learn-bootstrap-4-in-30-minute-by-building-a-landing-page-website-guide-for-beginners-f64e03833f33>

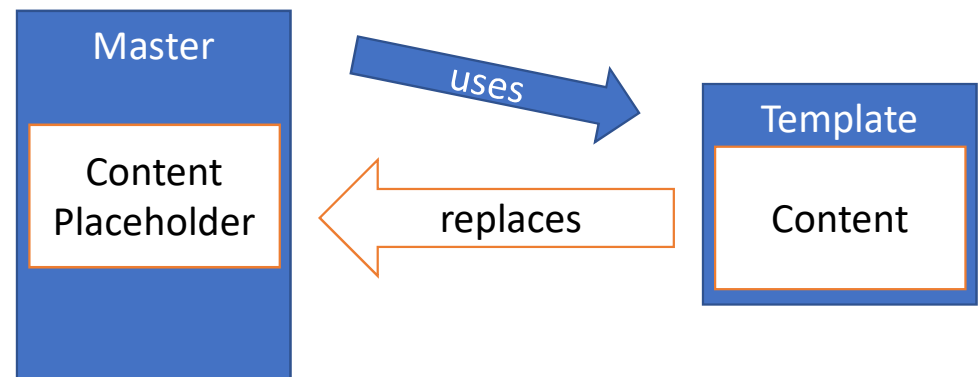
# Master Template – Template Composition

Django Templates support „Template Composition“ to combine similar template via a master template

- Define a master template with one ore more content placeholder blocks
- Use this master template with other templates and fill the placeholders with content

→ This results in: Every template has the same

- Style
- Header
- Footer
- Menu
- ...



# Master Template

- Contains the frame with all common parts over all templates
- And one or more placeholders  
{ % **block content** % }
- Save the master template under the name „base.html“

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>People</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
</head>
<body>
  <!-- Navigation Bar -->
  <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
    <!-- Brand/logo -->
    <a class="navbar-brand" href="{% url 'person_index_route' %}">People</a>
    <!-- Links -->
    <ul class="navbar-nav mr-auto">
      <li class="nav-item">
        <a href="{% url 'person_index_route' %}" class="btn btn-outline-light">All People</a>
      </li>
      <li class="nav-item">
        <a href="{% url 'person_create_route' %}" class="btn btn-outline-light">Create Person</a>
      </li>
    </ul>
  </nav>
  <!-- content placeholder -->
  {% block content %}
    if you see this, something is wrong!
  {% endblock content %}
  <!-- JavaScripts -->
  <script src="https://code.jquery.com/jquery-3.3.1.slim.js"
    integrity="sha256-hwg4gsxgFZzhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
    crossorigin="anonymous">
  </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
</body>
</html>
```

# Index View

- Load the master view with

```
{% extends 'base.html' %}
```
- Fill the placeholder section with

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

```
...
```

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

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

{% block content %}
<h1>All People</h1>
<table class="table table-striped table-bordered">
  <thead>
    <td>Firstname</td>
    <td>Lastname</td>
    <td colspan="3">Actions</td>
  </thead>
  {% for person in people %} {# repeat for every person in people #}
  <tr>
    <td>{{ person.firstname }}</td>
    <td>{{ person.lastname }}</td>
    <td>
      <a href="{% url 'person_detail_route' person.id %}"
        class="btn bg-light btn-outline-dark">Show Details</a>
    </td>
    <td>
      <a href="{% url 'person_edit_route' person.id %}"
        class="btn bg-light btn-outline-dark">Edit</a>
    </td>
    <td>
      <form method="post" action="{% url 'person_delete_route' person.id %}">
        {% csrf_token %}
        <button type="submit" class="btn bg-light btn-outline-dark">Delete</button>
      </form>
    </td>
  </tr>
  {% endfor %}
</table>
<p>
  Number of People: {{ people.count }}
</p>
{% endblock content %}
```

# The resulting solution

The screenshot shows a web browser window with the address bar at `localhost:8000/people/`. The page has a dark header with the title 'People' and two buttons: 'All People' (selected) and 'Create Person'. Below the header, the main content area is titled 'All People' and contains a table with three rows of people data. Each row has three buttons: 'Show Details', 'Edit', and 'Delete'. At the bottom of the page, it says 'Number of People: 3'.

Firstname	Lastname	Actions
Hugo	Hurtig	<button>Show Details</button> <button>Edit</button> <button>Delete</button>
Susi	Samtig	<button>Show Details</button> <button>Edit</button> <button>Delete</button>
Leo	Lump	<button>Show Details</button> <button>Edit</button> <button>Delete</button>

Number of People: 3