* Open PyCharm Professional -> File -> New Project
* python manage.py startapp tasks
* To include an app in a project, **add a reference** to the app in the **INSTALLED\_APPS** setting
* **Setting up a Database**

1. Startapp appname
2. Add to ‘installed app’
3. Create urls.py in the app
4. Include app`s urls.py in the project`s urls.py

**DJANGO INTRODUCTION**

**DJANGO ADMIN**

* Добавянето на първия потребител в Джанго админа (администратора)

Python manage.py createsuperuser 🡪 следваш стъпките

* Регистриране на модела в Джанго админа 🡪 във файла admin.py в самия app

@admin.register(Task)  
class TaskAdmin(admin.ModelAdmin):  
 pass

* Set **list\_display** to control which fields are displayed on the change list page of the admin.

@admin.register(Task)  
class TaskAdmin(admin.ModelAdmin):  
 list\_display = ('id', 'name', 'priority')

* **list\_filter** – може да сортираш по опфределен атрибут

list\_filter = ('email\_address',)  
search\_fields = ('first\_name', 'last\_name')

* **fields**– как да се визуализират
* **fieldset** – групират се по секции

fieldsets = [  
 ('Personal Info', {  
 'fields': ('first\_name', 'last\_name')  
 }  
 ),  
 (  
 'Professional Info', {  
 'fields': ('email\_address', 'birth\_date')  
 }  
 )  
]

**VIEWS** – приемат HTTP request и връщат response

1 view представлява 1 функция, която взима един единствен параметър – request

* За да се визуализира view-то тряябва да се добави в url-а на app-a / да се регистрира

urlpatterns = [  
 path('', index, name='index'),  
]

.objects - достъпваме мениджър

* Показване на всички данни от модела – във view.py

def get\_all\_tasks(request):  
 all\_tasks = Task.objects.all()

TEMPLATES

* Показване на template – във view.py

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

* Context – подаваме чрез него обекти от базата от данни или обекти, които искаме да се визуализират; dict , което подаваме от view-то и може да се визуализира в template

def index(request):  
 all\_tasks = Task.objects.all()  
 context = {  
 'title': 'The tasks app!',  
 'tasks': all\_tasks,  
 }  
 return render(request, 'index.html', context)

**VIEWS AND URLS**

* Dynamic Path Segments

В < > се поставя динамичния параметър

path('department/<department\_name>/', views.show\_department\_by\_name)

* Optionally, can include **converter type** (otherwise, it is converted to a string)

path('department/<int:department\_id>/', views.show\_department\_by\_id)

* The value name is passed as an **argument** to the **view**

def show\_department\_by\_id(request, department\_id):

...

* Типове
* **str** – matches any **non-empty string**, excluding "/"
* **int** – matches **zero** or any **positive** integer
* **slug** – matches any slug string consisting of ASCII **letters**, **numbers**, **hyphens**, and **underscores**

**REDIRECT**

def redirect\_to\_first\_department(request):  
 possible\_order\_by = ['name', 'age', 'id']  
 order\_by = choice(possible\_order\_by)  
 return redirect('show departments/name of path in urls.py')

\*Но трябва да си е направен и path с redirect/

* Грешки

def show\_not\_found(request):  
 status\_code = 400  
 # return HttpResponseNotFound('This is not found!')#First Var  
 return HttpResponse('Error', status=status\_code) #Second Var

raise Http404(`Not found`) #Third Var

DEBUG – True (за development)

DEBUG = True  
  
ALLOWED\_HOSTS = []

DEBUG – False (за production)

DEBUG = False  
  
ALLOWED\_HOSTS = [  
 'localhost',  
 '127.0.0.1',  
]

**Django Templates**

* DTL
* Променливи {{ }}
* Филтри/ Filters 🡪 <https://docs.djangoproject.com/en/4.2/ref/templates/builtins/#add>
* Тагове/Django tags (ф-я, която изпълняваме върху нашите данни) {% %} – ЗАТВАРЯТ СЕ 🡪 https://docs.djangoproject.com/en/4.2/ref/templates/builtins/

{% for е in employee %} {% endfor %}

* {% empty %} – ако няма нищо в employee
* {% ifchanged … %} {% endifchanged %} – ако се добави нещо ново – проверява дали текущата стойност е различна от предишната стойност
* {% url ‘url-a , към който искаме да отидем ’ %} – прави се навигация

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

* CSRF\_TOKEN – предпазва ни от фалшиви сайтове

{#{{ csrf\_token }}#} ВИЗУАЛИЗИРА СЕ  
{% csrf\_token %} СЪЗДАВА СЕ INPUT ПОЛЕ С VALUE CSRF\_TOKEN и type=”hidden”

* Comments – single line {# comment #}, multi line {% comment %} Here add comment {% endcomment %}
* Custom filters – взима 1 параметър

1. В app-a се прави package ‘templatetags’
2. Вътре в package-a се правят файлове ‘filters’
3. from django.template import Library  
     
   register = Library()  
     
     
   @register.filter('odd')  
   def get\_odd(values):  
    return [x for x in values if x % 2 == 1]
4. Трябва да се зареди в html-a 🡪 {% load filters %} filters е името на файла
5. Използва се в темплейта 🡪

<div>  
 Odd: {{ values|odd }} odd е името на филтъра  
</div>

* Custom tags – simple\_tag (връща винаги стринг) (when python logic needed)

1. В app-a се прави package ‘templatetags’
2. Вътре в package-a се правят файлове “tags”
3. from django.template import Library  
     
   from templates\_demo.web.views import Student  
     
   register = Library()  
     
     
   @register.simple\_tag(name='student\_info')  
   def show\_student\_info(student: Student):  
    return f'Hello, my name is {student.name}'
4. Зарежда се в html файла

{% load tags %}

1. Използва се в темплейта 🡪

{% student\_info student %}

* Custom tags – inclusion\_tag – може да взима много параметри (when python logic needed)

1. В app-a се прави package ‘templatetags’
2. Вътре в package-a се правят файлове “tags”
3. @register.inclusion\_tag('tags/nav.html', name='app\_nav')  
   def generate\_nav(\*args):  
    context = {  
    'url\_names': args  
    }  
     
    return context
4. Зарежда се в html файла
5. {% load tags %}
6. Използва се в темплейта 🡪
7. Tag Nav:  
   {% app\_nav 'index' 'redirect to home' %}

Takes\_context – можем ад получим достъп, с който view-то е било извикано

* Template inheritance

1. В templates може да се направи папка base, където да се сложи base.html
2. Взеимат се нещата, които искаме да се повтарят
3. Там, където искаме да стават различните неша се слага:

{% block page\_content %}  
{% endblock %}

1. За да се наследи базовия темплейт – най-отгоре в дъщерния файл

{% extends 'base/base.html' %} РЕЛАТИВЕН ПЪТ КЪМ БАЗОВИЯ ТЕМПЛЕЙТ

1. {% block page\_content %} Въвеждат се нещата, които искаме да има наследника/дъщерния {% endblock %}

* {% include ‘partials/footer.html’ %} – дава се релативен път към html файла, може да се използва при готов html фоотер примерно/ използват се, когато не ни е нужна python logic
* Static files – CSS, JS и изображения

1. Правим си папка в проекта – staticfiles
2. Вътре може да вкарваме нашите файлове (примерно css, js)
3. За да я заредим 🡪 settings

STATIC\_URL = 'static/'  
  
STATICFILES\_DIRS = (  
 # join(BASE\_DIR, 'staticfiles'), FIRST VAR  
 BASE\_DIR / 'staticfiles', SECOND VAR  
)

1. Зарежда се static

{% load static %}

1. Добавя се към html

Wrong Way

<link rel="stylesheet" href="/static/styles.css">

Right Way

{% load static %}

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

BASE\_DIR (променлива, която достъпва пътя на проекта)

**DJANGO MODELS**

Django ORM – Object Relational Mapping – Позволява ни да пишем пайтън код, който да се превежда до SQL

Всеки клас/модел се свързва директно с някоя таблица

Django raw sql log – добавя се логика към settings.py с debug цели

* Fields (колони) – клас атрибути
* Типове
  + CharField – има required параметър max\_length=
  + TextField – за голям текст
  + DateTimeField – optional параметър auto\_now= True(this will be automatically set on each save), auto\_now\_add=True(this will be automatically set on creation)
* Прехвърляне на кода към базата от данни:
  + Базата, към която се свързваме 🡪 DATABASES в settings.py
  + Миграции – следят как се променят данните ни във времето, правят се когато имаме промяна по структурата; пазят история

Python manage.py makemigrations – прави миграции

Python manage.py migrate – аплайва миграции

* Когато вече сме мигрирали определена структура на таблицата , но добавяме нова колоне, не може да бъде Not Null и да няма стойност, затова Django дава два варианта:

1. Избираме 1) и даваме default-на стойност, която да се запише в новата колона

It is impossible to add a non-nullable field 'years\_of\_experience' to employee without specifying a default. This is be

cause the database needs something to populate existing rows.

Please select a fix:

1) Provide a one-off default now (will be set on all existing rows with a null value for this column)

2) Quit and manually define a default value in models.py.

Please enter the default value as valid Python.

The datetime and django.utils.timezone modules are available, so it is possible to provide e.g. timezone.now as a value.

Type 'exit' to exit this prompt

>>> 1

Не трий миграции, които не са аплайнати

* def \_\_str\_\_ - показва как да изглеждат данните в базата
* reverse на миграции – python manage.py migrate employees/апп-а/ 002/миграцията, до която искаш да се изтрият/
* reverse all migrations - python manage.py migrate employees/апп-а/ zero
* Допълнителни конфигурации на поле
  + Unique=True (всички записи в тази колона трябва да са уникални)
  + Default=10 (Подава дифоултна стойност)
  + Null – можем ли да слагаме null стойности в базата, ако Null=False дава да се попълни формата, но гърми(реалната валидация в базата от данни)
  + Blank – не се изпълнява валидация за null за формата (използва се за form validation) , тоест blank=False значи, че задължително трябва да се поддаде стойност (във формата)
  + models.BooleanField()-дава само checkbox, models.BooleanField(null=True)-дава drop down
  + choices(()) – дава dropdown, през формата се избира „Senior“, в базата се записва „sr“

job\_level = models.CharField(  
 choices=(  
 ('jr', 'Junior'),  
 ('reg', 'Regular'),  
 ('sr', 'Senior'),  
 )  
)

* + verbose\_name – поддава се какво име да има във формата, само за визуални цели
  + editable =False (първоначално може да се заададе, но не може да се променя), по дифоулт е True
* РЕЛАЦИИ – models.ForeignKey()

ONE TO MANY (може да се използва за категории)

department = models.ForeignKey(  
 Department,/на къде иска да ходи/  
 on\_delete=models.CASCADE /какво да му е поведението при delete/  
)

Когато правим релации на по-късен етап, трябва да се правят с две миграции – 1вата създава другата таблица(в случая Department), добавя се нещо в таблицата (в Department), 2рата е тази с foreignkey

ON DELETE OPTIONS

* CASCADE – Изтрива ли се department, изтрий всички employee
* SET NULL – работи само ако NULL=True, изтрива ли се деп., сетни деп. на емпл. на null
* RESTRICT – не можеш да изтриеш деп., ако имаш човек вързан към него

MANY TO MANY

Правим си нов модел:

class Project(models.Model):  
 name = models.CharField(max\_length=30)  
 code\_name = models.CharField(  
 max\_length=10,  
 unique=True  
 )  
 deadline = models.DateField()

И в class Employee правим релацията към Project:

projects = models.ManyToManyField(Project)

trought=‘EmployeesProjects‘ – опционален параметър, слага се, когато ние си създаваме междинна таблица

ONE TO ONE

class AccessCard(models.Model):  
 employee = models.OneToOneField(  
 Employee,  
 on\_delete=models.CASCADE,  
 primary\_key=True,  
 )