Django Generic Views

created 03-04-2020.

It is this EasyK - Kahden esimerkki -projektin kautta ihmettelyä siitä, mitä Django tarjoaa.

Suurin osa web-sovelluksista on sillä lailla samanlaisia, että niissä esitetään jokin lista ja se listalta voidaan valita kohde tarkasteluun.

Twitter, Facebook, Instagram. Stadin ravintolat. HJKn Jalkkis.net pelaajaseuranta. WhatsApp. Listoja ja kohdesivuja. Suurin osa.

Django on kehittynyt tähän suuntaan, jotta kehittäjien ei tarvitse toistaa näitä samoja asioita. Kovinkaan moni ei taida näitä ominaisuuksia aktiivisesti käyttää, sillä hyvin harva opetusvideo Youtubessa (huom myös lista-detail) tämän huomioi. Vähän on tosi, tosi hyvää.

Tässä miten itse sen teen:

Ensin luonnollisesti perustetaan projekti ja sille App. PyCharm on siinä hyvä, voi suositella.

Alla esimerkissä App = projektit, pienellä alkukirjaimella siis.

Perustettuun App -sovellukseen luodaan sitten ao. tiedostot.

models.py

Tähän taulukuvaus tai kuvaukset. Joka tauluun elämää helpottamaan __str__ funktio. Se luo automaattisen hakuavaimen tauluun.

```
from django.db import models


class Projekti(models.Model):
    title = models.CharField(max_length=32, blank=True)
    shortdescription = models.TextField()
    description = models.TextField(blank=True)
    image = models.ImageField(blank=True)
    def __str__(self):
        return self.title
```

Tauluissa voi olla paljonkin kenttiä ja tauluja voi olla tietysti niin paljon kuin sovellus vaatii. Tämä esimerkki, joka voi hyvin toimia Templatena ‘aina’, ei suurta määrää tarvitse. Neljä kenttää on esitystarpeisiin mainio, ehkä ‘linkki’ -kenttä muualle nettiin tai muihin (omiin) sovelluksiin vielä kuuluisi olla. Mutta yksi taulu jossa yksi kenttä riittää alkuun. Tyyliin “listaa kodin kirjat”.

views.py

Seuraavaksi tehdään näkymät, siis varsinainen liiketoimintalogiikka. Tässä Django todella, todella loistaa. Olen jo tottunut painimaan ja laatimaan isojakin views-funktio ja myös luokka (class) kokonaisuuksia, mutta tämä django.views.generic on jotain toista.

```
from django.views import generic


from .models import Projekti
class IndexView(generic.ListView):
    model = Projekti

class DetailView(generic.DetailView):
    model = Projekti
```

Siinä kaikki.

urls.py

Polkuja eri sivuille tehdään tässä. Jälleen django loistaa. Koska näkymät on määritelty funktioiden(def) sijaan luokkina(class), niin hommahan helpottuu:

```
from django.urls import path
from . import views


app_name = 'projektit'
urlpatterns = [
    path('projektit/', views.IndexView.as_view(), name='index'),
    path('projektit/<int:pk>/', views.DetailView.as_view(), name='detail'),
]
```

Tässä “name” -tiedotkin ovat optionaalisia, ne tekevät linkityksen vaan selväjärkiseksi siis helposti luettavaksi.

<projekti>_list.html

Html on sitten se jota käyttäjä näkee. Vähimmillään tämä toiminnallisuus saataisiin näin:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</head>
<body>

{% for proj in object_list %}
        <a href="{% url 'projektit:detail' proj.id %}">
                <p>{{ proj.title }}<p>
        </a>
{% endfor %}

</body>
</html>
```

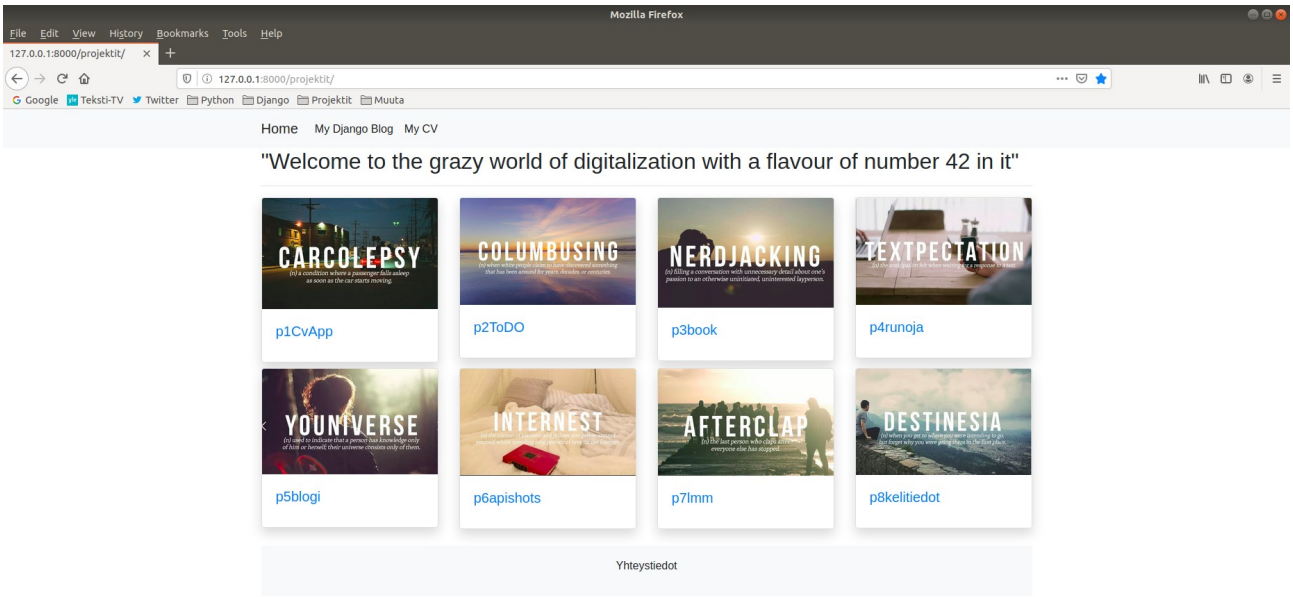
Listataan vain tietokannasta (models.py) löytyvät tiedot.

Koska kuitenkin halutaan käyttäjälle hyvä mieli, niin hommaan lisätään vähän nätteyttä Bootstrap -työvalineistön avulla. Sekin on helppoa kuin heinänteko, koska homma on visuaalista. Hyvin nopeasti löytää sen mitä haluaa. Ja Youtube ja muuta kanavat on täynnä apuja.

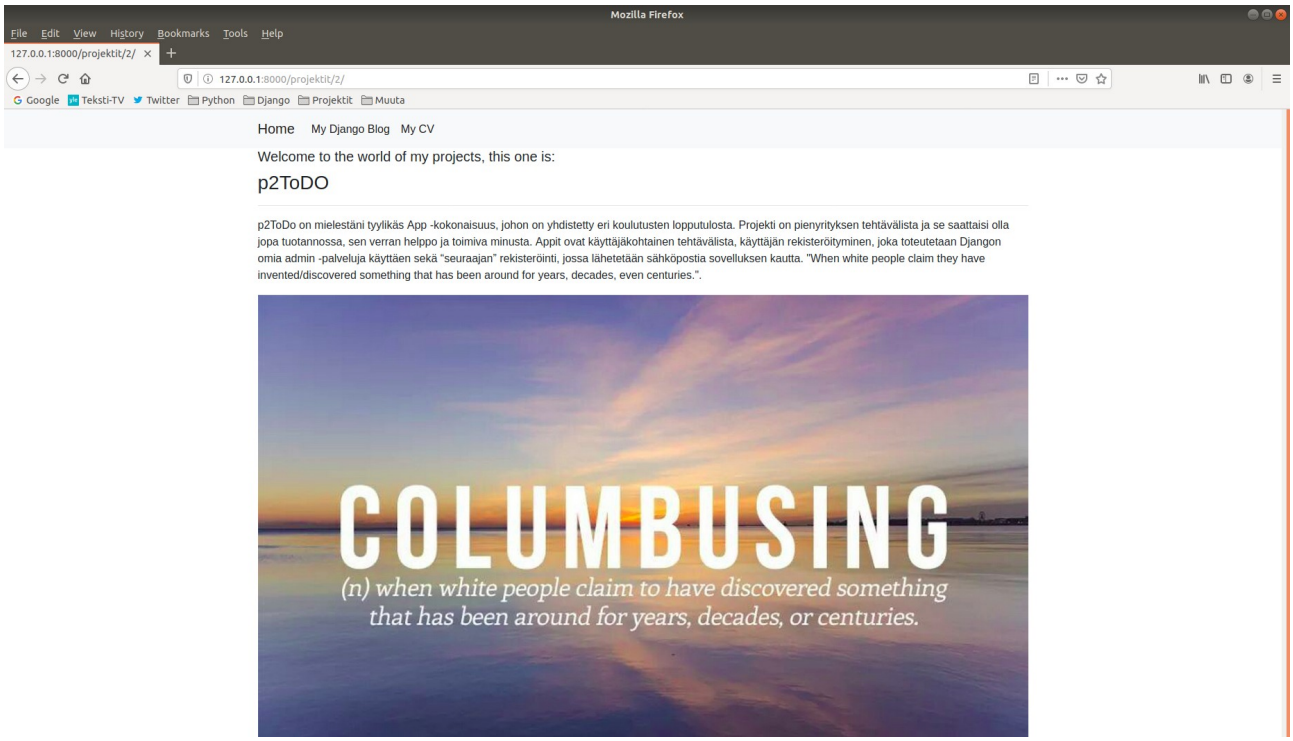
Minulla näin:

```
{% extends 'base.html' %}
{% block page_content %}
<script>
$(document).ready(function(){
  $('[data-toggle="tooltip"]').tooltip();
});
</script>
    <h2>"Welcome to the grazy world of digitalization with a flavour of number 42 in
it"</h2>
        <hr>
<div class="row">
{% load static %}
    {% for proj in object_list %}
        <div class="col-md-3">
            <a href="{% url 'projektit:detail' proj.id %}" data-toggle="tooltip"
title='{{ proj.shortdescription }}'>
            <div class="card mb-2 shadow">
                <img class="card-img-top" src="{% static 'images/' %}{{ proj.image }}">
                <div class="card-body">
                    <h5 class="card-title">{{ proj.title }}</h5>
                </div>
            </div>
            </a>
        </div>
    {% endfor %}
</div>
<footer class="container-fluid text-center p-3 my-3 navbar-light bg-light">
    <p>Yhteystiedot</p>
</footer>
{% endblock %}
```

Ei tämän kummemmin, mutta lopputulema on aika siisti, minusta ainakin:



ja <projekti>_detail.html



Alla vielä esimerkki p4RunoApp muokkaamisesta toimintopohjaisesta (function) käsittelystä luokkapohjaiseen enemmän objektiorientoituneeseen käsittelyyn (class). Ero kompleksisuudessa on jälleen merkittävä. Huomaa myös, että Poista-toimintoon on lisätty käyttäjän tunnistaminen.

Toimintopohjaisena ja vain vähäisellä geneerisellä näkymällä:

```
def runoLista(request):
    shelf = RunoDB.objects.order_by('-rate')
    return render(request, 'runoApp/runoListaweb.html', {'shelf': shelf})
def runoUusi(request):
    runouusi = RunoForm()
    if request.method == 'POST':
        runouusi = RunoForm(request.POST)
        if runouusi.is_valid():
            runouusi.save()
            return redirect('runoLista')
        else:
            return HttpResponse("""your form is wrong, reload on <a href = "{{ url :
'runoLista'}}">reload</a>""")
    else:
        context = {
            'runouusi': runouusi
        }
        return render(request, 'runoApp/runoUusiweb.html', context)
def runoKorjaa(request, runo_id):
    runo_id = int(runo_id)
    try:
        runo_sel = RunoDB.objects.get(id=runo_id)
    except RunoDB.DoesNotExist:
        return redirect('runoLista')
    runo_form = RunoForm(request.POST or None, instance = runo_sel)
    if runo_form.is_valid():
       runo_form.save()
       return redirect('runoLista')
    return render(request, 'runoApp/runoUusiweb.html', {'runoUusi':runo_form})
def runoPoista(request, runo_id):
    runo_id = int(runo_id)
    try:
        runo_sel = RunoDB.objects.get(id=runo_id)
    except RunoDB.DoesNotExist:
        return redirect('runoLista')
    runo_sel.delete()
    return redirect('runoLista')
```

Muunnettuna luokka-pohjaiseksi ja hyödyntäen Djangon geneerisiä ominaisuuksia:

```
class RunoList(ListView):
    model = RunoDB
class RunoRead(DetailView):
    model = RunoDB
class RunoCreate(CreateView):
    model = RunoDB
    fields = ['name',
              'caption',
              'author'
              ]
class RunoEdit(UpdateView):
    model = RunoDB
    fields = ['name',
              'caption',
              'author',
              'picture',
              'rate'
              ]
    template_name_suffix = '_update_form'
class RunoDelete(DeleteView):
    model = RunoDB
    success_url = reverse_lazy('runolista')
    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
```