Django, egy példán keresztül

Horák 'RePa' Gyuri, 2010

Az alapok

Ahogy korábban már írtam, régebb óta tervezem valami tutorial szerűség megírását, hát végre eljutottam ide - kicsit lassabban, mint terveztem, az utóbbi időszak nem várt eseményei miatt. A teljes tutorial több cikket fog magába foglalni, ez az első rész a django környezet kialakításáról fog szólni.

A folyamatot egy olyan példán keresztül szeretném bemutatni, amit később saját célra használni is fogok, néhány ponton ezért lehet, hogy nem a legegyszerűbb megoldásokat alkalmazom. A problémakört más irányból közelítem meg, mint a hivatalos django tutorial, ezért azt sem árt átnézni, illetve csak ajánlani tudom a django dokumentációját, ami szerintem kifejezetten jó.

A környezet kialakítása

Mielőtt rátérnénk a djangos környezet kialakítására, mindenkinek ajánlom figyelmébe Gábor barátom nemrégiben publikált Fejlesztői környezet kialakítása Python alapú fejlesztéshez című írását, én is nagyon hasonlóan szoktam eljárni minden új projektecske esetében. Ebben az esetben is első lépésként létrehoztam egy `djkeptar` nevű környezetet, majd ide installáltam a szükséges dolgokat:

```
$ mkvirtualenv djkeptar
$ workon djkeptar
$ pip install ipython django pysqlite
```

Az egyszerűség kedvéért `sqlite` adatbazist használtam, kisebb dolgokhoz és fejlesztéshez tökéletes, nem kell szenvedni *rendes* adatbázis szerverrel. Ezután a `django-admin.py` parancs segítségével hozhatjuk létre a django projektünket. Én szeretem minden kódomat verziókezelő rendszerben tárolni, így rendszeres commitolgatás mellett nem gond visszatérni valami korábbi állapotra, könnyen meg tudom osztani másokkal, és a modern DVCS-eknek hála több helyre is elküldhetem, hogy biztonságban tudjam. Én magam a Mercurial-t preferálom (python ugyebár), de ugyanolyan jó tapasztalataim vannak a `git`-tel is, ráadásul aki az egyiket megismeri, az a másikkal is el fog boldogulni.

```
$ django-admin.py startproject djkeptar
$ cd djkeptar
$ hg init
$ hg add
$ hg commit -m 'project letrehozasa'
```

A django egyik alapja a beilleszthető/újrafelhasználható alkalmazások (*pluggable/reusable apps*) rendszere, ami a DRY (*don't repeat yourself*) filozófiát követi, azaz ha már egyszer megcsináltál valamit (esetleg valaki más csinálta meg), akkor felesleges újra megcsinálni, használjuk azt, ami már kész, és valaki már valószínűleg lefutotta azokat a köröket vele, amit nekünk kéne, ha újat csinálnánk. Az interneten kb. végtelen ilyen django appot találunk, vannak közöttük egész jók, tehát mielőtt valami újba belekezdünk, mindenképp érdemes körbenézni - és ha számunkra tökéletes megoldás nem is született még, egy forkolásra megérett verziót nagy valószínűséggel találunk. (Például saját blogot mindenki írt/ír/írni fog djangoban. Tökéletest még nem láttam.)

Mivel minden lényegi munkát ezek az appok csinálnak, ezért bármit szeretnénk csinálni, készítenünk kell hozzá egy appot:

```
$ python manage.py startapp keptar
```

A parancs semmi mást nem csinál, csak létrehoz egy `keptar` nevű könyvtárat - ami egy python modul lesz egyébként - és a könyvtárban pár kvázi-üres python filet:

```
keptar/
    __init__.py
    models.py
    tests.py
    views.py
```

A file-ok nevei elég beszédesek, a `models.py` fogja tartalmazni az alkalmazásunk modell részét, a `views.py` a view-kat, a `tests.py` meg a teszteket. Az elnevezésekből már látszik, hogy django egy MVC-szerű framework, de mivel (szerintem) a webes világra a hagyományos MVC minta jelenleg csak kicsit erőltetve illeszthető rá, ezért ők ki is mondják, hogy csak valami hasonlót csinálnak.

Beállítások

A következő lépés a django projektünk konfigurálása, melyet a `settings.py` file-on keresztül lehet megtenni. (Illetve a témában az oldalon is született már néhány bejegyzés.)

Itt mindenképpen állítsuk be az adatbázishoz kapcsolódó paramétereket, illetve adjuk hozzá a készülő alkalmazásunkat és az admin alkalmazást az `INSTALLED_APPS` listához (csak a lényeg, ami nem maradt alapértelmezetten):

```
import os.path
PROJECT_DIR = os.path.dirname(__file__)
DATABASES = { 'default': {
  'ENGINE': 'django.db.backends.sqlite3',
  'NAME': os.path.join(PROJECT_DIR, 'keptar.sqlite'),
}}
MEDIA_ROOT = os.path.join(PROJECT_DIR, 'media')
TEMPLATE_DIRS = (
  os.path.join(PROJECT_DIR, 'templates'),
)
INSTALLED_APPS = (
  # nehany default, azokat ne bantsuk
  'django.contrib.admin',
  'keptar',
)
```

Amint látható egy apro trükköt alkalmaztam a direkt elérési úttal megadandó dolog helyének megállapításához, mégpedig hogy a file elején megállapítom az ő helyét a filerendszeren, és ehhez képest relatív módon adom meg a többit (pl. `MEDIA_ROOT` vagy `TEMPLATES_DIR`).

Szinte kész is vagyunk, az `urls.py` file-ban (*erről bővebben majd később*) engedélyezzük az admin elérését:

```
from django.conf.urls.defaults import *
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
```

```
  url(r'^admin/', include(admin.site.urls)),
)
```

Szinkronizáljuk az adatbázist az appjaink modelljeivel:

```
$ python manage.py syncdb
```

Erre azért van szükség, mert - bár mi magunk még nem készítettünk semmi olyat, aminek adatbázisban a helye - az admin felülethez, illetve a felhasználók kezeléséhez alapból tartoznak modellek. Első futtatáskor rá is kérdez az első *admin* felhasználó adataira. Később, ha új appot adunk a rendszerhez, vagy változik a modellünk(*), akkor a `syncdb` management parancs újbóli futtatása szinkronizálja a változásokat.

> (*): Ez azért sajnos nem ennyire egyszerű, ha egy már beszinkronizált modellünk sémája változik, azt az alap django nem tudja kezelni. Azonban erre is van megoldás, mégpedig a south, amit én előre látó módon a példa projektben el is helyeztem, de most nem szeretnék róla írni, mert külön cikket érdemel.

Ha minden jól ment, akkor a környezet létrehozásával kész is vagyunk, a fejlesztői szervert futtatva ellenőrizhetjük, hogy minden rendben működik-e:

```
$ python manage.py runserver
Validating models...
0 errors found

Django version 1.2.3, using settings 'djkeptar.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

A fejlesztői szervert más porton (illetve publikus IP címen) is elindíthatjuk, simán paraméterként `[[ip/host:]port]` módon megadva, pl:

```
$ python manage.py runserver djkeptar.hu:5000
```

Kész is vagyunk a környezet létrehozásával, sőt, a böngészőnkbe beírva a `http://localhost:8000/` URL-t egy szép 404-es hibaüzenet fogad minket, mellyel akkor találkozhatunk, ha a `settings.py`-ben a `DEBUG` változó értéke `True`. (Éles rendszeren ez szigorúan tilos!)

A hibaüzenetben látszik, hogy bár az üres (`/`) címen nincs semmi, de a `/admin/` címen van valami. Ezt megnézve a djangotól "*ingyen*" kapott admin felülettel találkozhatunk, ahol jelen pillanatban felhasználókat tudunk csak kb. létrehozni. Érdemes ismerkedni vele, nagyon hasznos dolog, fejlesztés kezdeti szakaszában tökéletesen használható, sőt van annyira flexibilis, hogy az esetek nagy részében sikerül a megrendelő kívánságainak megfelelően testre szabni, és így megspórolhatjuk egy teljesen új admin felület létrehozását.

Létrehoztunk tehát egy django projektet, ami már képes a futásra, használ adatbázist, felhasználókat kezel, de egyébként semmire nem jó :)

Látható, hogy még így management parancsokkal megtámogatva is sok olyan lépes van, amit minden egyes új projektünknél végre kéne hajtani - bár valószínűleg nem készítünk naponta többtíz ilyet, ezt mégis fel lehet picit gyorsítani, pl. ha egy közepesen felkonfigurált django projektet eltárolunk kedvenc verziókezelőnkben, majd azt vesszük alapul a következőknél. (pl. íme Gábor saját django-boilerplate-je)

View-k és template-ek

Djangoban a `view`-k felelnek meg nagyjából az *MVC minta* controllereinek. Tipikusan olyan függvények - vagy függvényként viselkedő objektumok -, amelyekhez hozzá van rendelve valamilyen URL-minta, és ha a felhasználó a böngészőjébe az adott mintának megfelelő URL-t ír be, akkor a `view` lefut, az általa visszaadott válasz (általában valami `HttpResponse` objektum) pedig a megfelelő formában visszajut a böngészőbe, és ott megjelenik a kívánt tartalom.

URL kezelés

Természetesen, hogy milyen URL-minta esetén milyen `view` fusson le, azt nekünk kell megadnunk, amit nagyon egyszerűen a projekt könyvtárában található `urls.py` fileban tehetünk meg. Az itt található `urlpatterns` listát kell bővítgetnünk `url(regularis kifejezes, view függvény [, egyéb opcionális paraméterek, például a view-nak átadandó argumentumok])` bejegyzésekkel. (Az `url` függvény meghívása helyett használhatunk sima python felsorolásokat (tuple), ez esetben a django maga hívná meg velük az `url` függvényt. Én szeretem kiírni.)

A reguláris kifejezés lesz a minta, amire illeszkednie kell az URL-nek, egyébként egy hagyományos python regexp (`r'minta'`), ami ha tartalmaz nevesített illesztéseket (pl. `(?P<postid>\d+)`), akkor azokat a view függvényünk paraméterként megkapja.

Az egyszerűség kedvéért a view függvény konkrét megadása helyett megadhatjuk csak a pontokkal elválasztott elérési útját, mint stringet (*dotted path*), sőt, ha valamelyik appunk rendelkezik saját `urls.py`-vel, arra itt hivatkozhatunk az `include(appneve.urls)` direktíva segítségével.

Lássunk egy példát az egészre (`urls.py`):

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
  url(r'^/?$', 'app.views.index'),
  url(r'^page/(?P<page_id>\d+)/(?P<slug>[\w-]+)/$', 'app.views.page'),
  url(r'^about/$', 'app.views.page', {'page_id': 1, 'slug': 'about'}),
  url(r'^admin/', include(admin.site.urls)),
)
```

A fenti példában a főoldal lekérésekor az `index` nevű view fut le, a `/page/2/valami/` meglátogatásakor a `page` nevű view hívódik meg `page_id=2` és `slug='valami'` argumentumokkal, a `/about/` hatására szintén a `page` fut le, de az előre megadott paraméterekkel, míg ha az URL `/admin/`-nal kezdődik, akkor az `admin.site.urls` modul szerint folytatódik a view-feloldás.

> Vannak, akik nem ilyen központosított módon szeretik tárolni az url-szabályaikat, hanem valahogy a view közelében szeretnék a mintát a view-hoz hozzárendelni. Kis trükkel erre is van lehetőség, például itt található egy dekorátoros megoldás.

A view-k

Ahogy fentebb írtam, a view-k djangoban egyszerű függvények. Első bemenő paraméterük kötelezően egy `HttpRequest` objektum - ezen keresztül jutnak hozzá a GET, POST, session és egyéb hasonló dolgokhoz -, illetve egy HttpResponse objektumot adnak vissza futás után. Nézzünk erre egy egyszerű példát:

```
from django.http import HttpResponse

def index(request):
  return HttpResponse('Helló világ!')
```

Ez így szuper egyszerű, viszont igen rondán nézne ki, ha komplett HTML oldalakat írnánk meg szövegként a view-inkon belül, ezért ezt a megoldást elég ritkán alkalmazzuk. Helyette template-eket használunk, azokban írjuk le a visszaadni kívánt adatok megjelenését, a view-kban ezeket a template-eket renderelejük visszaadható állapotba.

Egyébként a sima `HttpResponse` mellett a django rendelkezik még ennek speciális leszármazottaival, melyekkel egyszerűen tudunk szabványos módon átirányítani (`HttpResponseRedirect`), vagy hibaüzeneteket visszaadni (`HttpResponseForbidden`, `HttpResponseNotFound`). A *Not found (404)* hibaüzenetet egyébként a Http404 exception (*kivétel*) dobásával is elérhetjük, ez sok esetben kényelmesebb.

Egy példa a template használatra:

```
# views.py
from django.template import Context, loader
from django.http import HttpResponse

def hello(request, name='Látogató'):
  if name = 'Sanyi':
    # Sanyit nem szeretjuk, neki nem koszonunk
    return HttpResponseForbidden('Utállak...')

  t = loader.get_template('hello.html')
  c = Context({
    'name': name
  })

  return HttpResponse(t.render(c))
```

```
{# hello.html #}
<html>
<h1>Hello kedves {{ name }}!</h1>
<p>Hogy vagy?</p>
</html>
```

A rendereléshez szükséges megadni a kontextust, egy `Context` objektum formájában, ami kb. egy python *dict*-et tartalmaz, ennek segítségével adhatunk át adatokat a tempalte-nek. Mivel sok app igényli, hogy `request` is elérhető legyen a template-ből, ezért én sima `Context` helyett `RequestContext`-et szoktam használni, ami ugyan olyan, csak második argumentumként meg kell neki adni a `request` objektumot.

Az utolsó néhány művelet a legtöbb esetben mindig ugyanígy szerepelne a view függvényeinkben, ezért a djangos srácok csináltak rá egy wrapper függvényt, hogy egyszerűsítsék a dolgokat, íme az előző view tömörebben:

```
# views.py
from django.shortcuts import render_to_response

def hello(request, name='Látogató'):

  return render_to_response('hello.html', {'name': name})
```

A tempalte-ek

A django template nyelve nem fog sok meglepetést okozni azoknak, akik használtak már valamilyen template nyelvet. A vezérlési szerkezeteket `{%` és `%}` közé kell tenni, a változók értékét `{{ valami }}` módon írathatjuk ki, illetve megjegyzéseket is írhatunk hasonló módon: `{# megjegyzés #}`. A template-ekben blokkokat definiálhatunk, leszármazhatunk belőlük - és a leszármazottban felüldefiniálhatjuk a blokkokat, illetve saját template-tageket is tudunk készíteni.

A `settings.py` fileban a `TEMPLATE_DIRS` listában tudjuk megadni, hogy a django hol keresse a tempalte-eket, emellett a django még benéz a telepített app-ok `templates` könyvtárába is, ha valamit nem talál az általunk megadott helyeken.

Vissza a képtárhoz

Első lépésben amolyan file-browsert akartam csinálni a képtárhoz. Egy (`settings.py`-ben megadott) könyvtár tartalmát böngészhetné a felhasználó, és az itt található képeket nézhetné meg. Csináltam pár általánosabb - djangotól független - függvényt, amiket az `utils.py` modulban helyeztem el a `keptar` appon belül.

> Az `utils.py` tartalma nem témája a tutorialnak, de nyugodtan bele lehet nézni, fileok és könyvtárak listázásra, thumbnail készítésére, és egyéb hasonló dolgokra találhatók benne függvények.

Két view-t definiáltam ebben a lépésben, az egyik egy konkrét kép, a másik pedig egy könyvtár tartalmának megjelenítésére képes (csak a lényeg):

```
# views.py
from keptar.utils import get_filelist, get_abspath, get_parent, enrich

def listdir(request, path=""):

    try:
        files = get_filelist(path)
    except:
        return HttpResponseForbidden('Access Forbidden')

    return render_to_response('listdir.html', {
        'path':     path,
        'parent':   get_parent(path),
        'files':    files,
        }, context_instance = RequestContext(request))

def showfile(request, fname):

    try:
        abspath = get_abspath(fname)
        fdata = enrich([fname])[fname]
    except:
        return HttpResponseForbidden('Access Forbidden')

    return render_to_response('showfile.html', {
        'parent': get_parent(fname),
        'fname': fname,
        'fdata': fdata,
        }, context_instance = RequestContext(request))
```

```
{# base.html #}
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>{% block 'title' %}Keptar{% endblock %}</title>
  <link rel="stylesheet" href="/media/css/style.css"/>
</head>
<body>
  <div id="container">
    <div id="main">
    {% block 'main' %}
    {% endblock %}
    </div>
  </div>
</body>
</html>
```

```
{# listdir.html #}
{% extends 'base.html' %}
{% block 'main' %}
<h1>{{ path }}</h1>
<a href="{% url listdir parent %}">parent{% if parent %} ({{ parent }}){% endif %}</a>

<ul>
{% for fname,fdata in files.items %}
  <li><a href="{{ fdata.url }}"><img alt="{{ fname }}" src="{{ fdata.thumb }}"/> {{ fname }}</a></li>
{% endfor %}
</ul>
{% endblock %}
```

```
{# showfile.html #}
{% extends 'base.html' %}
{% block 'main' %}
<h1>{{ fname }}</h1>
<a href="{% url listdir parent %}">parent{% if parent %} ({{ parent }}){% endif %}</a>

<div>
  <img alt="{{ fname }}" src="{{ fdata.direct_url }}"/>
</div>
{% endblock %}
```

Amint látható maguk a viewk nem túl bonyolultak, az `utils.py` függvényei segítségével lekértük a file/könyvtár listát, illetve a kép adatokat (*amit jelen esetben felfoghatunk modellnek is*), majd az adatokkal lerendereltettük a megfelelő template-et.

Az `if` és a `for` tempalte-tageket nem magyaráznám, ellenben említést érdemel az `url` tag, ami `{% url viewneve param1 param2 %}` módon visszaadja az adott view adott paraméterezéséhez tartozó URL-t az érvényben lévő `urls.py` alapján. Ha már szóba került, vegyük fel bele az új view-kat:

```
urlpatterns = patterns('',
  url(r'^/?$', 'keptar.views.listdir'),
  url(r'^list/(?P<path>.*)$', 'keptar.views.listdir', name='listdir'),
  url(r'^show/(?P<fname>.*)$', 'keptar.views.showfile', name='showfile'),
  url(r'^admin/', include(admin.site.urls)),
)
```

A `base.html` template-ben hivatkozok külső stíluslapra is (`style.css`), amit a django is ki tud szolgálni statikus tartalomként, ehhez fel kell venni az url minták közé az alábbi sort:

```
url(r'media/(?P<path>.*)$', 'django.views.static.serve', {'document_root': settings.MEDIA_ROOT}),
```

Természetesen éles üzemben ezt nagyon nem javaslom, a webszerver maga sokkal gyorsabban tud kiszolgálni statikus fileokat, mint a django.

Az extra `name` paraméterrel hivatkozhatunk a szabályunkra rövid névvel az `{% url %}` tagben.

Ahogy említettem az általam használt paramétereket is a `settings.py`-ben kell beállítani, így nem kell a felhasználóknak valami extra fileban is turkálniuk, ha az alkalmazásomat ők is használni szeretnék:

```
KEPTAR_ROOT='/var/www/foto'
KEPTAR_URL='http://dyuri.horak.hu/foto/'
KEPTAR_EXTENSIONS=['jpg','jpeg','png']
KEPTAR_THUMBDIR='.tn'
KEPTAR_THUMBSIZE=(120,120)
KEPTAR_SHOW_HIDDEN=False
KEPTAR_ICONS={
  'dir': 'http://dyuri.horak.hu/keptar/icons/tn_dir.jpg',
}
```

A kódból ezeket a változókat egyébként az alábbi módon érhetjük el:

```
from django.conf import settings
valami = settings.KEPTAR_ROOT
```

Elvileg kész is vagyunk, a fejlesztői szervert futtatva (`python manage.py runserver`) böngészhetjük is a `settings.KEPTAR_ROOT` könyvtár tartalmát.

A modell

Egy MVC jellegű alkalmazás legfontosabb része a modell, a legelső dolog, amit meg kell terveznünk, el kell készítenünk. (Bár én a végére hagytam, vegyük észre, hogy eddig is volt modell a példában, mégpedig az `utils.py` modulon keresztül elért filerendszer.)

A modellünket természetesen nem írja meg helyettünk a django, de rendelkezik egy elég jó ORM-mel, segít a validációban, és ugye van egy automatikusan generált admin felülete, ahol végül is a modellünket piszkálhatjuk.

A modellünket - nem meglepő módon - az appunk `models.py` moduljában kell definiálnunk. Semmi trükköset nem kell elképzelni, hagyományos python osztályokat kell létrehoznunk, annyi megkötéssel, hogy az osztályoknak a `django.db.models.Model` osztályból kell származniuk, illetve az adatbázisban eltárolandó mezőket előre definiálnunk kell.

A képárba szerettem volna egy foto-blog szerű funkciót, ami annyit tesz, hogy a filerendszer böngészése közben megjelölhetnék képeket, amik aztán a megjelölés dátumának sorrendjében jelennének meg a "*blogban*". A bejegyzéseknek szeretnék egy címet és címkéket adni.

Mivel a django alapból nem rendelkezik címke mezővel, ezért két lehetőségünk van: vagy mi magunk csinálunk valami hasonlót, vagy keresünk egy kész megoldást django app formájában, és azt használjuk. Én utóbbi mellett döntöttem - főként azért, hogy megmutassam hogyan kell egy külső django appot használni a projektünkben -, és Alex Gaynor django-taggit nevű munkáját választottam. Ahhoz, hogy elérhetővé váljon a

modelljeink számára telepíteni kell (`pip install django-taggit`) és hozzáadni a `settings.py` modulunk `INSTALLED_APPS` listájához. Ha ezzel kész vagyunk, készítsük el a modell definíciónkat a `keptar` appunk `models.py` moduljában:

```
class PBlogEntry(models.Model):

    path = models.CharField(max_length=1000, unique=True)
    title = models.CharField(max_length=200)
    user = models.ForeignKey(User)
    mark_date = models.DateTimeField('date marked', auto_now_add=True)
    tags = TaggableManager()

    class Meta:
        verbose_name = 'PBlog bejegyzes'
        verbose_name_plural = 'PBlog bejegyzesek'

    def is_valid(self):
        """ellenorzi, hogy a 'path' utvonalon levo file letezik-e"""
        abspath = get_abspath(self.path)
        return os.path.isfile(abspath)

    @property
    def fdata(self):
        """a fizikai filehoz tartozo adatok"""
        return enrich([self.path])[self.path]

    def __unicode__(self):
        return u"%s (%s)" % (self.title, self.path)
```

Első körben a modell mezőit definiáljuk: - `path`: A kép elérési útja, ahogy listázáskor is hivatkozunk rá. Egyedi, azaz egy képet csak egyszer jelölhetünk meg blogbejegyzésnek. - `title`: A bejegyzésünk címe. - `user`: A felhasználó, aki megjelölte a képet. A `ForeignKey` adattípuson keresztül hivatkozhatunk más modell-objektumokra. - `mark_date`: A megjelölés dátuma. Az `auto_now_add` paraméter miatt ezt a django majd automatikusan kitölti nekünk. - `tags`: Az előbb installált `django-taggit` *varázs* mezője, ami a címkézést végzi.

Az adatmezőkön kívül más dolgok is helyet kaptak az osztályban, ne felejtsük el a modell nem csak a perzisztencia réteget, de az üzleti logikát is jelenti (bár jelen példát igen erős túlzás üzleti logikának nevezni :), annyit szerettem volna mondani ezzel, hogy nyugodt szívvel használjunk itt értelmes metódusokat, és ne a view-inkban manipulálgassuk a modell-objektumainkat valami varázs függvényekkel):

- `is_valid`: Ellenőrzi, hogy a kép fizikailag megtalálható-e.
- `fdata`: Egy olyan property, ami a fizikai filehoz tartozó infókat adja vissza.
- `__unicode__`: Az objektum `unicode` reprezentációja, amikor valahol `unicode`-dá (illetve stringgé) kell konvertálni az objektumot, akkor ez hívódik meg. A legegyszerűbb ilyen eset a `print objektum` parancs.

Természetesen attól az adatbázisunkba nem kerül bele az új modell sémája, mert beleírtuk a `models.py` fileba, ki kell ehhez adnunk pár parancsot:

```
$ python manage.py validate
0 errors found
$ python manage.py syncdb
...
```

Ha mindezzel kész vagyunk, akkor parancssorból ki is tudjuk próbálni a modellünket. Ehhez a django szintén nyújt támogatást a `shell` management parancs formájában:

```
>>> from keptar.models import PBlogEntry
>>> from django.contrib.auth.models import User
>>> import datetime
>>> e = PBlogEntry(path='proba/kep.jpg', title='Proba Kep', mark_date=datetime.datetime.now(), user=User.objects.get())
>>> e.save() # ez menti el az adatbazisba
>>> e.tags.add('proba')
>>> e.tags.add('kep')
>>> masike = PBlogEntry.objects.get() # mar az adatbazisbol szerdjuk ki a legelso PBlogEntry objektumot
>>> masike.tags.all()
[<Tag: proba>, <Tag: kep>]
>>> print masike
Proba Kep (proba/kep.jpg)
```

Ha az admin felületen is látni és piszkálni szeretnénk a modellünket, akkor ahhoz regisztrálnunk kell őt az admin appnál. Ezt a regisztációt elvileg bárhol megtehetnénk - pl. magában a `models.py`-ben is, de érdemes az appunk gyökerébe egy `admin.py` nevű fileban megtenni, az admin app ezeket behúzza. A regisztráció maga egy `admin.site.register(PBlogEntry)` paranccsal megoldható lenne, de lehetőségünk van kicsit testre szabni az admin által generált listát, illetve formot, például:

```
from keptar.models import PBlogEntry
from django.contrib import admin

class PBlogEntryAdmin(admin.ModelAdmin):

    # a listaban metodusokat is szerepeltethetunk
    list_display = ('path', 'title', 'user', 'mark_date', 'is_valid')
    search_fields = ['path', 'title']
    date_hierarchy = 'mark_date'
    list_filter = ['user']

admin.site.register(PBlogEntry, PBlogEntryAdmin)
```

Ha kész vagyunk, a fejlesztői szervert elindítva már láthatjuk is az új modellünket az admin oldalon, sőt az előbb felvett `proba/kep.jpg` elemünk is megvan, amiről látszik is a listában, hogy nem valid (javaslom ezért a törlését, mert csak bekavar később).

Űrlapok

Gondolhatnánk, hogy ha már a django az admin felületre képes a modelleinkhez űrlapokat generálni, akkor miért ne lenne képes erre a műveletre a mi kérésünkre. És valóban, amellett, hogy a `django.forms` modul elemeiből kézzel készítenénk űrlapokat, az adminos mókához hasonlóan a modellekhez képes a django is űrlapot generálni (én ezeket szintén külön, a `forms.py` modulba szoktam elhelyezni):

```
from django import forms
from keptar.models import PBlogEntry

class PBlogEntryForm(forms.ModelForm):
    class Meta:
        model = PBlogEntry
        exclude = ('user')
        widgets = {
            'path': forms.HiddenInput(),
        }
```

Amint látható annyi a dolgunk, hogy leszármazunk a `django.forms.ModelForm` osztályból, és a belső `Meta` osztályon belül mondhatjuk meg, hogy pl. melyik modellhez szeretnénk űrlapot (`model = ModelClass`), illetve közölhetünk olyan dolgokat még a generátorral, hogy milyen mezők maradjanak ki (`exclude`), vagy hogy ha valami mezőt nem az alapértelmezett widgettel (*ez mi magyarul? építőelem?*) szeretnénk megjeleníteni (`widgets`).

A `shell` management parancs segítségével meg is nézhetjük, hogy hogyan néz ki a generált formunk, egyszerűen annyit kell tennünk, hogy példányosítjuk az osztályt:

```
>>> from keptar.forms import PBlogEntryForm
>>> f = PBlogEntryForm()
>>> print f.as_p() # ha siman kiiratjuk, akkor tr/td elemeket hasznal, amit en nem szeretek
<p><label for="id_title">Title:</label> <input id="id_title" type="text" name="title" maxlength="200" /></p>
<p><label for="id_tags">Tags:</label> <input type="text" name="tags" id="id_tags" /> A comma-separated list of tags.
<input type="hidden" name="path" id="id_path" /></p>
```

Tehát ahhoz, hogy a form megjelenjen az oldalunkon, elég annyit tennünk, hogy a view-nkban létrehozunk egy `PBlogEntryForm` objektumot, ennek esetleg adunk néhány alapértelmezett értéket (pl. `path`), ezt átadjuk a context objektumon keresztül a template-nek, ahol egy `<form>` tag-en belül kiíratjuk.

Helyezzük el hát az űrlapot a kép nézet oldalon, közvetlen a kép fölött, csak akkor, ha belépett felhasználó nézi az oldalt. Ehhez módosítani kell a `showfile` nevű view-nkat:

```
# reszlet a keptar/views.py filebol
def showfile(request, fname):

    try:
        abspath = get_abspath(fname)
        fdata = enrich([fname])[fname]
    except:
        return HttpResponseForbidden('Access Forbidden')

    # ha be van lepve valaki, akkor beteheti a kepet a photoblogba
    if request.user.is_authenticated:
        try:
            # ha az elem mar szerepel az adatbazisban, akkor a formban az o
            # adatait szeretnenk latni
            form = PBlogEntryForm(instance=PBlogEntry.objects.get(path=fname))
        except PBlogEntry.DoesNotExist:
            # ha nem szerepel, akkor uj, ures formot szeretnenk
            form = PBlogEntryForm(initial={'path': fname})
    else:
        # nincs belepve senki, nem kell urlap
        form = None

    return render_to_response('showfile.html', {
        'pbform': form, # a template-nek pbform neven adjuk at az urlapot
        'parent': get_parent(fname),
        'fname': fname,
        'fdata': fdata,
        }, context_instance = RequestContext(request))
```

Illetve a `templates/showfile.html` template-ünkben is helyezzük el az űrlapot:

> Annyit tennék még hozzá, hogy a django alapértelmezetten bekapcsolt CSRF védelemmel érkezik, azaz a `settings.py` modulban a `MIDDLEWARE_CLASSES` listában szerepel a `django.middleware.csrf.CsrfViewMiddleware` osztály. Ezesetben az összes űrlapunknál használni kell a `{% csrf_token %}` template-taget, ami egy `hidden` mezőben tartalmazni fogja a felhasználó biztonsági kódját, ami nélkül az űrlap érvénytelen.

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

{% block 'main' %}
<h1>{{ fname }}</h1>
<a href="{% url listdir parent %}">parent{% if parent %} ({{ parent }}){% endif %}</a>

{% if pbform %}
<form action="{% url submitpbentry %}" method="post">
  {% csrf_token %}
  {{ pbform.as_p }}
  <p><input type="submit" name="submitpbe" value="Ok" /></p>
</form>
{% endif %}

<div>
  <img alt="{{ fname }}" src="{{ fdata.direct_url }}"/>
</div>
{% endblock %}
```

A `form action` paraméterének egy külön view-t adtam meg, ami az űrlap hibáinak kezelése szempontjából (*amit a django szintén ügyesen támogat, de most nem foglalkoznék vele*) nem feltétlen előnyös, viszont én szeretem külön tudni a form kezelő view-kat, leválasztva őket a tisztán megjelenítésért felelő részekről (kicsit controller-view jellegű szétválasztás, de ne erőltessük), illetve így könnyebb az űrlapot több különböző oldalon is használni. Az űrlap feldolgozó view ilyenkor egy átirányítással továbbítja a célhelyre a böngészőt, aminek az az előnye is megvan, hogy a böngésző *újratöltés* gombjának hatására nem küldi el újra az űrlapot. Lássuk hát az űrlap feldolgozó view-nkat:

```
def submitpbentry(request):

  # ha nincs belepve, akkor nem szabad
  if not request.user.is_authenticated:
      return HttpResponseForbidden('Access Forbidden')
  try:
      # ha az adott kep mar szerepel az adatbazisban, akkor az o adatait szeretnenk frissiteni
      f = PBlogEntryForm(request.POST, instance=PBlogEntry.objects.get(path=request.POST['path']))
  except PBlogEntry.DoesNotExist:
      # ha nem szerepel, akkor uj elemet hozunk letre a form alapjan
      f = PBlogEntryForm(request.POST)

  # ha a felhasznalot nem raknank hozza, akkor siman menthetnkenk,
  # igy viszont kulon kell menteni a kapcsolodo adatokat is (tag)
  pbe = f.save(commit=False)
  pbe.user = request.user
  pbe.save()
  # kapcsolodo adatok (tag-ek) mentese
  f.save_m2m()

  return HttpResponseRedirect(reverse('showfile', args=[pbe.path]))
```

Kicsit bonyolultabb eset ez annál, amivel kezdeni kellene (érdemes megnézni a jóval egyszerűbb hivatalos django tutorial ide vonatkozó részét), de jó példa arra, hogy adhatunk a beérkező űrlaphoz olyan adatokat, amit nem szeretnénk semmiképp a felhasználóra bízni. Természetesen az urljeink közé is fel kell venni az új view-t, amit az `urls.py` modulban az alábbi módon tehetünk meg:

```
urlpatterns = patterns('',
    # sokminden ...
    url(r'^submitpbe$', 'keptar.views.submitpbentry', name='submitpbentry'),
)
```

Ezek után bőszen jelölgethetjük a képeinket, amiket utána az admin oldalon szerkeszthetünk is. Már csak egy új view/tempalte párosra van szükségünk, hogy blog szerűen nézegethessük a megjelölt képeket:

```
# keptar/views.py

  def pblog(request, id=None, slug=None):

    try:
        # ha az id nincs megadva, akkor a legutolsot jelenitjuk meg
        if id is None:
            pbe = PBlogEntry.objects.latest('mark_date')
        else:
            pbe = PBlogEntry.objects.get(pk=id)
    except PBlogEntry.DoesNotExist:
        # hibas id volt megadva, vagy nincs meg bejegyzes
        return render_to_response('pblog.html',
            {},
            context_instance = RequestContext(request))

    # elozo es kovetkezo elem meghatarozasa idorendi sorrendben
    next = PBlogEntry.objects.filter(mark_date__gt=pbe.mark_date).order_by('mark_date')[:1]
    # python 2.6+ eseten ez sokkal szebb lenne:
    # next = next[0] if len(next) > 0 else None
    if next:
        next = next[0]
    prev = PBlogEntry.objects.filter(mark_date__lt=pbe.mark_date).order_by('-mark_date')[:1]
    if prev:
        prev = prev[0]

    return render_to_response('pblog.html', {
        'pbe': pbe,
        'next': next,
        'prev': prev,
        }, context_instance = RequestContext(request))
```

```
{# templates/pblog.html #}
{% extends 'base.html' %}

{% block 'main' %}
{% if pbe %}
<h1>{{ pbe.title }}</h1>
<div class="nav">
  {% if prev %}<a href="{% url pblog prev.id prev.title|slugify %}">Previous</a>{% endif %}
  <a href="{% url showfile pbe.path %}">Browse</a>
  {% if next %}<a href="{% url pblog next.id next.title|slugify %}">Next</a>{% endif %}
</div>

<h2 class="tags">Tags:
  {% for tag in pbe.tags.all %}
  <span class="tag">{{ tag }}</span>{% if not forloop.last %}, {% endif %}
  {% endfor %}
</h2>

<div>
  <img alt="{{ pbe.title }}" src="{{ pbe.fdata.direct_url }}"/>
</div>
{% else %}
<h1>The photoblog is still empty...</h1>
{% endif %}
{% endblock %}
```

```
# urls.py reszlet
urlpatterns = patterns('',
```

```
    url(r'^pblog/(?P<id>\d+)/(?P<slug>[\w-]*)/$', 'keptar.views.pblog', name='pblog'),
    url(r'^pblog/(?P<id>\d+)/$', 'keptar.views.pblog'),
    url(r'^/?$', 'keptar.views.pblog'),
    # ...
)
```

A sima `/pblog/<id>/` és a főoldal (`/`) url-én kívül legelső helyen egy olyan mintát adtam meg, ami az azonosító után egy *slugot* vár, ami egy betűkből, számokból és kötőjelből álló valami. Ezt is letárolhatnám a modellemben, de értelme leginkább azért van, hogy *szebbek* legyenek az url-ek, az ilyesmit a keresők is jobb helyre szokták sorolni, és az emberek is szívesebben kattintanak rá. Ezt a slugot a template-en belül a `sulgify` szűrővel készíthetjük el.

Elindítjuk, és **örülünk**!

> Időközben rájöttem, hogy finoman szólva buta dolog, hogy a `settings.py`-be beledrótoztam a saját gépemen lévő képek elérési útját, ezért elnézést kérek, a relimgdir mercurial címke alatt elérhető az a verzió, ahol azt átalakítottam egy relatív eléréssé, ami a projekt `images` könyvtára, illetve ide el is helyeztem két képet. Szóval, ha valaki csak úgy letölti és elindítja, akkor ez a verzió jó eséllyel produkál valami értelmes eredményt.

Konklúzió

Mit is ad nekünk a django? Egy MVC jellegű keretrendszert, modell oldalon okos ORM támogatással, egy jól használható, kibővíthető template nyelvet, űrlap kezelést, URL routingot, middleware (*köztesréteg-modul? omg*) rendszert, egy meglepően jól használható automatikusan generált admin felületet, illetve ami szerintem a legnagyobb ereje - főként új project induláskor -, azok a könnyedén beépíthető alkalmazások rendszere. Mindemellett nagy előny, hogy nem köti meg a kezünket, a felsorolt dolgok közül semmit sem kötelező használnunk. Nem tetszik a template nyelv? Sebaj, használhatunk bármi más (python) template nyelvet, pl. Jinjat. Nem tetszik az űrlap kezelés (ez mondjuk meglepne), csinálhatunk sajátot, vagy ott van a WTForms. Az ORM a szűk keresztmetszet? Hát nem kötelező használni, használhatunk tisztán SQL-t, vagy akár valami nem relációs adatbázist is, pl. ott a mongoengine (*ez esetben mondjuk az automatikusan generált admin felülettől is el kell búcsúznunk*).

Próbáltam minél teljesebb képet adni, mégis most úgy érzem, hogy csak a felszínt karcolgattam. De hát ha nem lenne már miről írni, akkor az oldalra sem lenne szükség tovább :) Mindenkinek örömteli ismerkedést kívánok a djangoval, és ha kérdésetek van, ne tartsátok magatokban! Természetesen a projekt messze nincs még kész, ha időm engedi folytatni fogom, és ha valami érdekeset csinálok, akkor arról megpróbálok beszámolni. Egyébként meg az egész fent van a bitbucketen, szabad forkolni, és szívesen veszem a *pull-requesteket* :)

A cikksorozat részei:

- Django, egy példán keresztül I. - Az alapok
- Django, egy példán keresztül II. - View-k és template-ek
- Django, egy példán keresztül III. - A modell

A teljes projekt szabadon elérhető a bitbucketen.