

Błyskawiczny kurs Django
Projekt 'równodzień'
obsługa systemu bibliotecznego

Żeby rozpocząć pisanie aplikacji w frameworku Django należy najpierw otworzyć linię poleceń, przejść do katalogu w którym zamierzamy trzymać kod źródłowy i wydać polecenie:

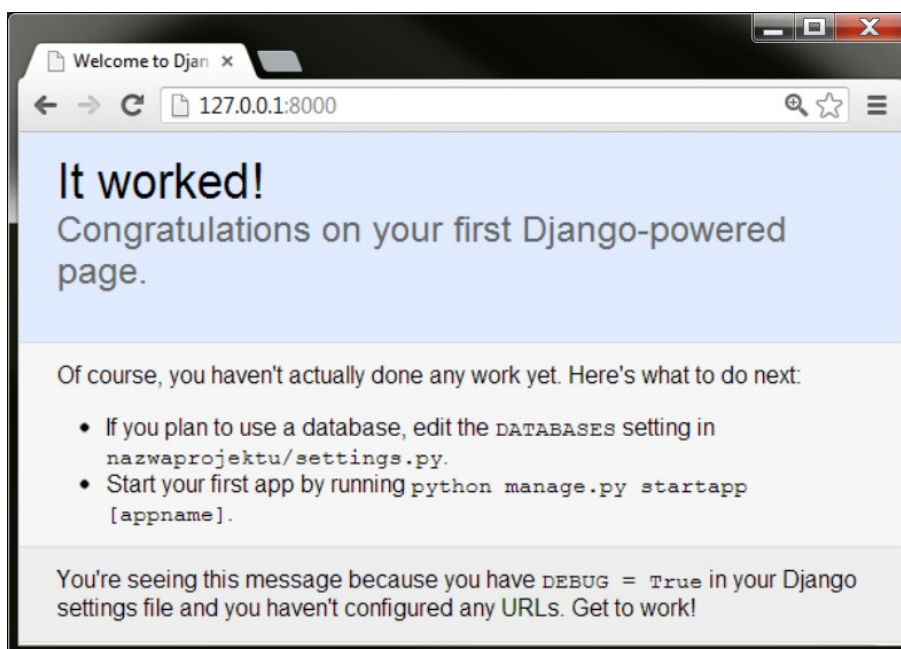
```
c:\python27\python c:\python27\scripts\django-admin.py startproject nazwap
```

Oczywiście ścieżki mogą się różnić w zależności od tego gdzie Python jest zainstalowany, jednak przy domyślnych ustawieniach polecenie brzmi tak. Zamiast nazwap podstawiamy nazwę naszej aplikacji - która musi być poprawnym identyfikatorem Pythona¹.

Po chwili mamy gotową podstawę aplikacji. Wejdźmy teraz do jej katalogu i wydajmy polecenie zgodnie z poniższym screenem:

```
D:\projects>cd nazwaprojektu  
D:\projects\nazwaprojektu>c:\python27\python manage.py runserver  
Validating models...  
  
0 errors found  
Django version 1.4.1, using settings 'nazwaprojektu.settings'  
Development server is running at http://127.0.0.1:8000/  
Quit the server with CTRL-BREAK.
```

Widzimy że do celów tworzenia aplikacji nie musimy instalować żadnego serwera - Django ma już swój zintegrowany. Wchodzimy na zadaną stronę:



Jeśli to widzimy to znaczy że Django działa. Otwórzmy sobie teraz katalog z projektem, celem omówienia konkretnych plików.

Katalog ten zawiera plik `manage.py` oraz podkatalog o takiej samej nazwie. Plik `manage.py` pozwala nam wydawać polecenia - takie jak `runserver`. Tych poleceń jest więcej i mają różne zastosowania - kilka z nich poznamy później. Wchodząc w podkatalog widzimy inne pliki. `__init__.py` to plik - może być nawet pusty - że ten katalog jest prawidłowym modulem Pythona. `settings.py` to plik z

¹ <http://docs.python.org/release/2.5.2/ref/identifiers.html>

danymi konfiguracyjnymi - za chwilę będziemy go edytować. `wsgi.py` pozwala na instalację projektu Django na profesjonalnych serwerach takich jak Apache - na razie nas nie interesuje. `urls.py` zawiera informację dla Django jak URL (adres wprowadzany w przeglądarce) przetłumaczyć na procedurę Pythona która go obsłuży.

UWAGA! Jeśli znasz PHP, pewnie będzie to dla ciebie zdziwieniem że URL-ami nie są nazwy plików z parametrami. Wybranie sposobu takiego jakim posługuje się Django (tzw. pretty URL) skutkuje większą przejrzystością strony, a także możliwością szybkiego ustalenia co w aplikacji jest czym.

Sam plik `settings.py` może na początku przerażać. Znajdują się w nim nastawy które informują Django co do używanych baz danych, strefy czasowej czy nawet języka domyślnego (Django wspiera tworzenie aplikacji wielojęzycznych). Na samym początku interesuje nas instalacja bazy danych. Zakładamy że mamy już zainstalowany PostgreSQL i `psycopg2`. Nastawy będą wyglądać mniej więcej tak:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'baza',
        'USER': 'postgres',
        'PASSWORD': 'jakieshaslo',
        'HOST': '',
        'PORT': '',
    }
}
```

Będą się różnić w kwestii hasła i utworzonej bazy danych dla Postgresa. O tym, czy to zadziała, dowiemy się dopiero później.

Aplikacje i kontrolery

Django zbudowane jest w ten sposób aby pomagać programiście w szybkim "ponownym wykorzystywaniu" kodu. Na ile mu to wychodzi znacznie uzależnione jest od programisty, ale póki co naszym celem nie jest nauka inżynierii oprogramowania. Chcemy zrobić pierwszą, prostą stronę w Django.

Najpierw więc musimy wiedzieć że projekt Django trzyma swoje kody w tzw. aplikacjach. Aplikacje to logiczne podziały projektu - mając aplikację-blog możemy na przykład mieć aplikację *użytkownicy*, *posty* czy *administracja*². Rozumiemy że do pewnego stopnia będą się one przenikać - nic nie szkodzi aby jedna aplikacja korzystała z kodu drugiej. Nazwy aplikacji muszą być poprawnymi identyfikatorami Pythona - dobrze jest aby nie kłóciły się też z nazwami jego wbudowanych modułów. Dobrze jest więc nazywać je po polsku, jeśli jeszcze niezbyt dobrze znamy Pythona.

Stwórzmy więc sobie pierwszą aplikację. Będąc w folderze projektu, wywołujemy polecenie zgodnie z zrzutem:

```
D:\projects\nazwaprojektu>c:\python27\python manage.py startapp pierwsza
```

² Choć Django jest w stanie za nas wygenerować panel administracyjny...

Nasza aplikacja tutaj nazywa się *pierwsza*. Jak widzimy, tworzymy ją za pomocą `manage.py`. Jak widzimy, utworzony został teraz dodatkowy folder³. Widzimy znajomy `__init__.py`. W pliku `models.py` będziemy umieszczać za chwilę tzw. modele - które będą reprezentować tabele w bazie danych. `tests.py` będzie zawierał testy jednostkowe - które także nie są przedmiotem tego kursu. Możemy `tests.py` bezpiecznie skasować, lub zostawić. W `views.py` możemy umieszczać nasze kontrolery - choć jeśli tego chcemy, można stworzyć też inne pliki Pythona na wzór `views.py` i umieszczać w nich kontrolery. Aplikację dodatkowo należy zarejestrować w pliku `settings.py`, modyfikując go zgodnie z zrzutem:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pierwsza',
    # Uncomment the next line to enable the admin:
```

Otwórzmy teraz `views.py` i wprowadźmy do niego następujący kod:

```
# coding=UTF-8
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse('Hello world')
```

`#coding=UTF-8` - jeśli jest obecny - musi być pierwszą linią w pliku. Oznacza że ten plik będzie w kodowaniu UTF-8 - i tak powinien być zapisany przez edytor. Python nie próbuje zgadywać w jakim kodowaniu są pliki które przetwarza. Można pominąć tą linię - jednak zemści się to przy najbliższym wpisywaniu polskich liter w kod. Umieszczanie tej linii na początku i zapisywanie pliku w formacie UTF-8 jest dobrym nawykiem programistycznym.

Drugą linią jest poinformowanie Pythona że będziemy korzystać z klasy `HttpResponse` z podmodułu `http`, należącego do modułu `django`. Django oczekuje że funkcja którą wywoła w celu obsłużenia naszego wywołania zwróci właśnie taką klasę.

Po linii odstępu mamy nagłówek. Jest ona jasna - definiujemy tutaj funkcję o nazwie `hello_world`, pobierającą jeden parametr - `request`. Obiekt `request` będzie zawierał informacje o przeróżnych rzeczach które życzył sobie użytkownik i nie tylko - chociażby zmienne GET i POST⁴ przekazywane do skryptu. Funkcja ta jest kontrolerem (w sensie MVC) naszego wywołania.

³ Do pewnego stopnia rozmieszczenie tych folderów można kontrolować i nie ma ono istotnego znaczenia dopóki poprawnie określimy ścieżki w `settings.py`

⁴ [http://pl.wikipedia.org/wiki/POST_\(metoda\)](http://pl.wikipedia.org/wiki/POST_(metoda))

Ostatnia linia zwraca nowo utworzoną klasę *HttpResponse*. Jeśli argumentem jej konstruktora jest ciąg znaków - a tak robimy tutaj - to tekst ten będzie zwrócony użytkownikowi który skorzysta z tego wywołania. Zobaczmy je w akcji później.

W tej chwili mamy funkcję która coś robi, ale zupełnie nie wiemy jak się do niej dostać. Aby umożliwić do niej dostęp, musimy powiązać ją z konkretnym URL-em w pliku `urls.py`. Dodajemy jedną linię, tak aby wyglądał podobnie jak poniżej:

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
    url(r'^hello/', 'pierwsza.views.hello_world'),
)
```

Wpis ten definiuje że URL-owi *hello/* przypisana jest funkcja *hello_world* znajdująca się w module *view*, który to jest w module *pierwsza*.

UWAGA! Python operuje pojęciami modułów i podmodułów w odniesieniu do plików i katalogów. Zauważ że to co wpisujemy jako ścieżka do *hello_world* znajduje odzwierciedlenie w strukturze plików i katalogów - a katalog który jest modulem zawiera w sobie plik `__init__.py`.

Zapis `^hello/` to wyrażenie regularne. `^` informuje nas że URL musi się od tego zaczynać. Umiejętność wpisywania wyrażeń regularnych pozwala na jeszcze ciekawsze manipulacje URL-ami.

Należy uruchomić teraz serwer (*runserver*) i wejść na `http://localhost:8000/hello/`. Powinien pojawić się napis **hello world**. Pojawia się jednak pewien problem - to nie jest kod HTML, a jedynie taki ciąg znaków, co można zobaczyć w *Pokaż źródło* przeglądarki. Ciężko jednocześnie wpisywać cały kod HTML bezpośredni w funkcji-kontrolerze. Jak poradzić sobie z tym problemem, dowiemy się potem.

Szablony

Szablon, jak sama nazwa wskazuje, to pewien wzór do wypełnienia. Będziemy tak nazywać pliki HTML z pewnymi "lukami", wprowadzonymi przez specjalne kody Django. Stworzymy w folderze *pierwsza* podfolder *templates*. Będziemy w nim umieszczać szablony związane z tą właśnie aplikacją, choć nic nie stoi na przeszkodzie aby umieszczać je we wspólnym folderze *templates*. Zanim szablony zaczną działać poprawnie, musimy poinformować Django gdzie należy ich szukać.

UWAGA! Ponieważ Django skorzysta z pierwszego pliku który ma zgodną nazwę i zostanie znaleziony, czasem efektywne jest stworzenie jednego katalogu *templates* z którego będzie korzystać wiele aplikacji. Unikamy wtedy niejednoznaczności i możliwości pomyłki.

W tym celu ponownie otworzymy plik *settings.py* i wprowadzimy następujący dodatek do zmiennej `TEMPLATE_DIRS` (po uprzednim sprawdzeniu i poprawieniu ścieżek):

```
TEMPLATE_DIRS = (  
    'D:/projects/nazwaprojektu/pierwsza/templates/',  
)
```

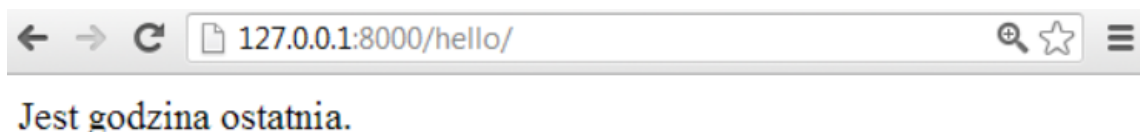
Nasz katalog z szablonami jest już zainicjowany. Teraz musimy utworzyć jakiś szablon. Stworzymy prostą stronę która będzie wyświetlać aktualną godzinę. Tak więc w folderze templates stwórzmy plik *godzina.html*. Będzie on wyglądał tak:

```
<!DOCTYPE HTML>  
<html>  
<head>  
    <title>Test</title>  
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
</head>  
<body>  
    Jest godzina {{ godzina }}.  
</body>  
</html>
```

Notacja `{{ godzina }}` oznacza że za ten tekst podstawiona będzie zmienna *godzina* którą z poziomu funkcji-kontrolera prześlemy do tego szablonu. Teraz, naszą funkcję *hello_world* zmodyfikujemy tak, aby korzystała z tego szablonu. Otwórzmy ponownie odpowiedni plik *views.py* i sprawdźmy aby funkcja wyglądała tak:

```
# coding=UTF-8  
from django.shortcuts import render_to_response  
  
def hello_world(request):  
    return render_to_response('godzina.html', {'godzina': 'ostatnia'})
```

O co tu właściwie chodzi? Funkcja *render_to_response* zwraca obiekt typu *HttpResponse* (dokładnie taki jakiego zwrócenia oczekuje od nas Django) zbudowany na podstawie szablonu o podanej nazwie i o podanych zmiennych. Zmienne podajemy szablonowi poprzez strukturę danych Pythona typu *dictionary* (słownik), gdzie kluczem jest nazwa zmiennej a wartością - oczywiście wartość zmiennej. Zapiszmy teraz projekt, uruchommy *runserver* i zobaczmy wynik:



Jest godzina ostatnia.

Wszystko dobrze, ale ten sam efekt moglibyśmy osiągnąć za pomocą zwykłego pliku HTML! Spróbujmy teraz czegoś trudniejszego - dynamicznego wygenerowania daty i czasu. Tym razem zmieniamy tylko funkcję-kontroler:

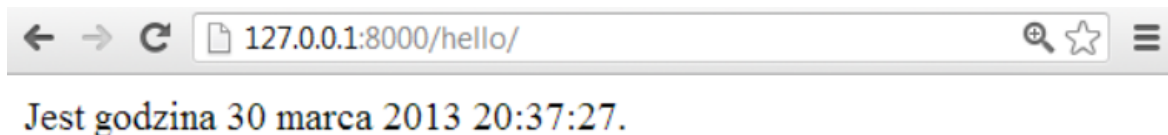
```
# coding=UTF-8
from django.shortcuts import render_to_response
from datetime import datetime

def hello_world(request):
    data = datetime.now()
    return render_to_response('godzina.html', {'godzina': data})
```

Po ponownym uruchomieniu *runserver* sprawdzamy wynik. Jest już nieco lepszy. Spróbujmy teraz ustawić polski tekst. W *settings.py* ustawmy *LANGUAGE_CODE* z *en-us* na *pl*.

PORADA Nie zawsze trzeba restartować *runserver*. W większości przypadków samo potrafi ono wykryć zmiany.

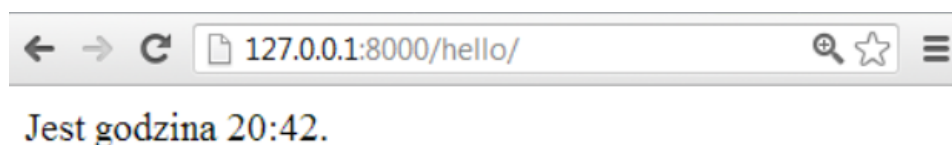
Efekt końcowy powinien wyglądać tak:



Kontroler kontroluje jakie dane zostają przekazane do wyświetlania, ale to szablon powinien kontrolować ich ostateczne formatowanie. Taka idea przyświeca właśnie szablonom Django. Dostajemy do szablonu datę, ale chcemy ją sformatować po swojemu. Do tego służą właśnie filtry. Otwieramy *godzina.html* i poprawiamy ją do takiej postaci:

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Test</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    Jest godzina {{ godzina|date:"G:i" }}.
</body>
</html>
```

Więcej o formatowaniu za pomocą daty można przeczytać pod <https://docs.djangoproject.com/en/dev/ref/templates/builtins/#date>. Ostateczny wynik będzie wyglądać tak:



Może wydawać się to sporym zachodem w porównaniu do PHP, jednak Django nie jest stworzone do wyświetlania daty! Można je zastosować do wielu bardziej skomplikowanych rzeczy. Teraz nauczymy się korzystać z potężnego systemu ORM⁵ który posłuży nam do dostępu do bazy danych.

Nasza strona WWW będzie księgą gości. W tym celu musimy stworzyć jedną tabelę, która będzie zawierać następujące pola (wszystkie wymagane).

1. Treść - typu tekstowego o nieograniczonej długości
2. E-mail dodającego - typu tekstowego o długości 256 znaków (maksymalnej długości e-mail)
3. Nick dodającego - typu tekstowego o długości 30 znaków
4. Kiedy dodano - typu data/czas

Warto przeczytać dokumentację Django o tym co za chwilę będziemy robić. Istotne są zwłaszcza artykuły <https://docs.djangoproject.com/en/dev/topics/db/models/> (wprowadzenie do tematu) oraz <https://docs.djangoproject.com/en/dev/ref/models/fields/> (lista typów).

Wprowadźmy do *models.py* aplikacji *pierwsza* taki oto kod:

```
# coding=UTF-8
from django.db import models

class Wpis(models.Model):
    tresc = models.TextField(verbose_name=u'Treść')
    email = models.EmailField(max_length=256, verbose_name=u'E-mail')
    nick = models.CharField(max_length=30, verbose_name=u'Nick')
    kiedy = models.DateTimeField(auto_now_add=True, verbose_name=u'Kiedy dodano')
```

Definiujemy tutaj klasę *Wpis*. Obowiązkowi musi ona dziedziczyć po klasie *Model* z modułu *models*. *verbose_name* służy do opisu "czytelnego dla człowieka", aby Django mogło potem automatycznie wygenerować formularz. Zapis *u'Treść'* informuje Pythona że chodzi nam o łańcuch znaków Unicode - mogący zawierać znaki narodowe. Zwykły *string* to po prostu ciąg bajtów. Co do reszty atrybutów - można zapoznać się z dokumentacją, zwłaszcza dotyczącą listy typów.

Teraz - zakładając że skonfigurowaliśmy już bazę danych - wydamy nowe polecenie za pomocą *manage.py*. Będzie to wyglądać tak:

```
D:\projects\nazwaprojektu>c:\python27\python manage.py syncdb
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table pierwsza_wpis

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): no
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

⁵ Object-Relational Mapper

Nie należy się przejmować dodatkowymi tabelami - w *settings.py* wymieniono oprócz naszej dużo dodatkowych, domyślnych aplikacji Django. Wspierają one dodatkowe funkcje, jak logowanie się, grupy użytkowników i pozwolenia - można je usunąć. Nie będziemy ich wykorzystywać, ale nie przeszkadzają nam też w pracy. *syncdb* jest niezbędne by zainicjalizować bazę danych odpowiednimi tabelami.

Otwierając teraz bazę danych możemy zobaczyć nowo utworzone tabele. Zauważmy że dzięki systemowi ORM Django dało się stworzyć tabelę w bazie SQL bez konieczności znajomości tego języka!

UWAGA! Znajomość SQL jest niezbędna by zrozumieć bardziej zaawansowane mechanizmy ORM Django oraz umiejętności pisanie wydajnych zapytań.

Dodatkowo, Django stworzyło pole podstawowe o nazwie *id* dla tabeli, mimo że nie zostało ono określone bezpośrednio w modelu. Dodamy teraz przykładowy wpis do tabeli. Skorzystamy w tym celu jeszcze z innej komendy Django - *shell* - oraz interaktywnego interpretera Pythona. Będzie to wyglądać tak:

```
D:\projects\nazwaprojektu>c:\python27\python manage.py shell
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from pierwsza.models import Wpis
>>> w = Wpis(tresc=u'Witaj!', email='test@example.com', nick='Testowy')
>>> w.save()
>>> exit()

D:\projects\nazwaprojektu>_
```

Zapis ten jest jasny. Ponieważ w momencie utworzenia obiekt w typie *Wpis* jeszcze nie jest zapisany w bazie musimy to zrobić jego metodą *save()*. Funkcja *exit()* powoduje opuszczenie interpretera Pythona. Żeby mieć dostęp do obiektu *Wpis* musimy go najpierw zaimportować - co dokonuje się w pierwszej linii.

Mamy teraz wpis. Dobrze byłoby teraz dodać możliwość wypisania wszystkich wpisów. Stwórzmy dodatkową funkcję-kontroler *wypisz()* - w pliku *views.py* naszej aplikacji. Plik po edycji wyglądał będzie tak:

```
# coding=UTF-8
from django.shortcuts import render_to_response
from datetime import datetime

def hello_world(request):
    data = datetime.now()
    return render_to_response('godzina.html', {'godzina': data})

from pierwsza.models import Wpis

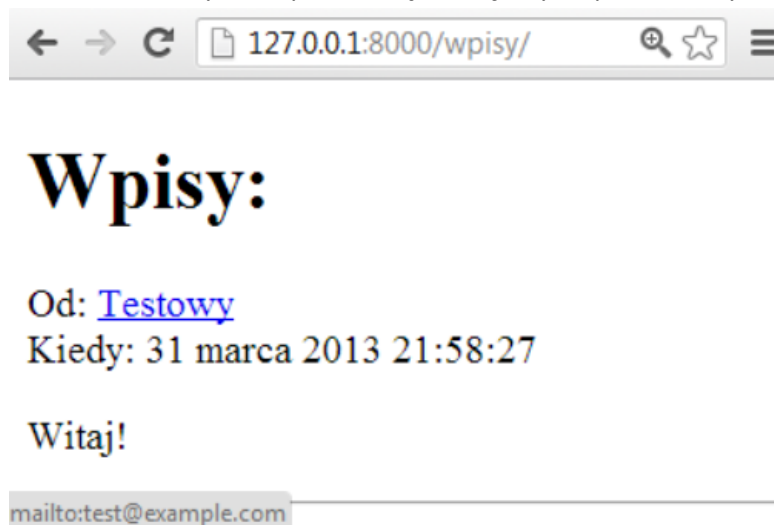
def wszystkie(request):
    wpisy = Wpis.objects.all()
    return render_to_response('wszystkie.html', {'wpisy': wpisy})
```

Operacje wykonywane na klasie *Wpis* tłumaczy dokumentacja - wstęp do modeli. Należy utworzyć teraz odpowiedni plik szablonu, z kodem:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Wpisy</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <h1>Wpisy:</h1>
  {% for wpis in wpisy %}
    Od: <a href="mailto:{{ wpis.email }}">{{ wpis.nick }}</a> <br>
    Kiedy: {{ wpis.kiedy }}<br>
    <p>{{ wpis.tresc }}</p>
    <hr>
  {% endfor %}
</body>
</html>
```

Tematykę takiej a nie innej składni dobrze tłumaczy

<https://docs.djangoproject.com/en/dev/ref/templates/api/>. Należy oczywiście teraz połączyć URL - powiedzmy *wpisy/* z tą funkcją w pliku *urls.py*. Robimy to analogicznie jak wcześniej. Po uruchomieniu *runserver* wchodzimy na odpowiednią stronę. Wynik powinien być taki:



Tak więc funkcja listy jest gotowa. Teraz należy stworzyć jakiś system który umożliwi dodawanie wpisów. Skorzystamy w tym celu z formularzy. Temat formularzy porusza <https://docs.djangoproject.com/en/dev/topics/forms/>. Z jego lektury wynika że musimy utworzyć wszystkie pola formularza osobno..

Oczywiście nie ma to najmniejszego sensu. Skorzystamy z funkcji Django omówionej w <https://docs.djangoproject.com/en/dev/topics/forms/modelforms/> - dzięki temu Django jest w stanie stworzyć formularz na podstawie gotowego już modelu. Zobaczmy jak działa to w akcji

Dodajmy teraz do naszego *views.py* kod:

```

from django import forms
from django.shortcuts import redirect

class DodajWpis(forms.ModelForm):
    class Meta:
        model = Wpis
        exclude = ['kiedy']

    def dodaj(request):
        if request.method == 'POST':
            form = DodajWpis(request.POST)
            if form.is_valid():
                form.instance.save()
                return redirect('/wpisy/')
        else:
            form = DodajWpis()

        return render_to_response('dodaj.html', {'form': form})

```

Omówimy chwilę w jaki sposób działa ten formularz. Klasa formularza - *DodajWpis*, dziedzicząca po *ModelForm*, dzięki czemu ma wcześniej opisaną funkcjonalność (konstrukcja formularza na podstawie modelu). Dodatkowy parametr - *exclude* - informuje formularz że nie powinien wyświetlać pola *kiedy*. Jest to zasadne, bo ustawiliśmy je wcześniej - przy specyfikacji modelu - żeby automatycznie się wypełniało datą bieżącą! Parametrem *model* w podklasie *Meta* (takiego zapisu spodziewa się *ModelForm*) jest nazwa (oczywiście wcześniej zaimportowanej klasy *Wpis*) klasy modelu na podstawie której chcemy wygenerować nasz formularz - tutaj jest to *Wpis*. W kodzie kontrolera, najpierw sprawdzamy czy dane rzeczywiście przesłano metodą POST. Jeśli tak (świadczy to o tym że użytkownik wypełnił formularz i wcisnął guzik wysyłania) to skonstruujemy ten formularz na podstawie tych danych - dzięki czemu będą w nim te dane. Następnie metodą *is_valid()* sprawdzimy czy są poprawne. Jeśli tak - to wtedy *form.instance* będzie klasą typu *Wpis* i będziemy mogli ją zapisać (o metodzie *save()* mowa była wcześniej), a następnie przejść do listy wpisów. Jeśli nie jest to zapytanie typu POST to tworzymy pusty formularz. W końcu do *render_to_response* trafi albo pusty formularz, albo wypełniony niewłaściwie formularz. Zostanie on wyświetlony odpowiednio przez szablon.

Wygląd szablonu formularza będzie prosty. Django potrafi wygenerować nie tylko obsługę, ale i kod HTML. Tak więc będzie on wyglądał tak:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Dodaj wpis</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <form action="/dodaj/" method="post">
    {{ form.as_p }}
    <input type="submit" value="Dodaj">
  </form>
</body>
</html>
```

Zapis `form.as_p` powoduje wywołanie metody `as_p()` tegoż formularza - której efektem działania jest zwrócenie reprezentacji formularza w postaci HTML, zawartej w akapitach (*p*, czyli *paragraph*). Niestety, trzeba samemu wpisać metodę HTTP oraz URL pod który mają być wysłane wyniki - ze względu na to że zarówno pierwszą generację formularza jak i późniejsze jego przetwarzanie realizujemy tym samym kontrolerem, wybieramy taki sam URL.

Następnie należy odpowiednio połączyć URL - w tym wypadku `dodaj/` - ze stosowną funkcją. Możemy następnie wejść na `http://127.0.0.1:8000/dodaj/` po uruchomieniu `runserver` i cieszyć się prostą księgą gości.

PORADA Do celów testowych można skorzystać z bazy SQLite. Dzięki temu w trakcie pisania programu nie musimy mieć zainstalowanej bazy danych i dopiero podczas wgrywania programu docelowo możemy po prostu zmienić `settings.py`.

UWAGA! Podczas przetwarzania formularza może wystąpić błąd CSRF. Należy wtedy w `settings.py` usunąć linijkę `'django.middleware.csrf.CsrfViewMiddleware'`,