Python and Django - The elephant in the room (Part 2)

Janosch Maier <maierj@in.tum.de>

April 09, 2015

Contents

1 Introduction

This tutorial let you build a scorecard application. Seminars and lectures can be added and then voted up or down. An overview page shows the courses ranked by the votes. This tutorial was build for the TUM seminar Webtech within the summer term 2015 [1]. The scorecard is put in a seperate directory as its own django application. This is useful if you want to have a broad range of functionality on your webpage and seperate their source code propperly. The final application is shown in figure 1.

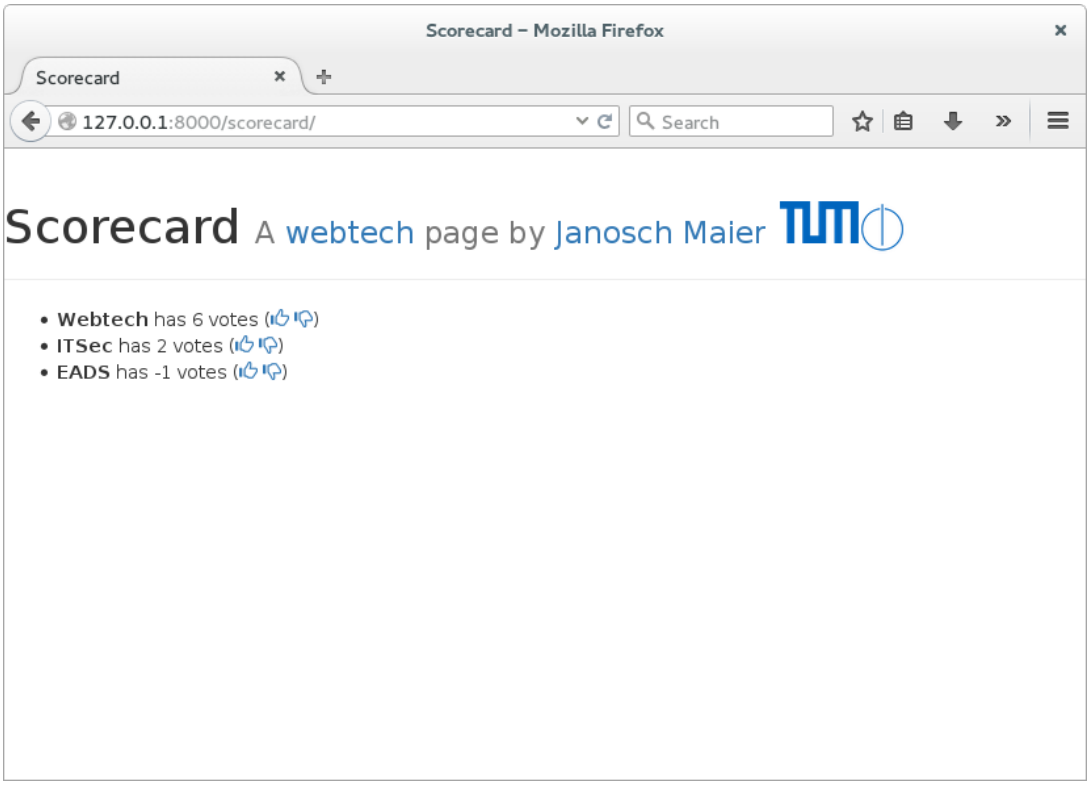

Figure 1: The scorecard Application

1.1 Requirements

To go through this tutorial you will need an installation of Python 3 with Django 1.7. You will probably get Python by your package manager. For help installing Django you should check out its documentation [2]. If you encounter any problems during this tutorial, refer back to that page. There is a lot of further information there.

1.2 Setup

You can either clone the application from GitHub or create it yourself.

1.2.1 Clone

To get the application from the GitHub use:

```
git clone https://github.com/Phylu/webtech-django2.git
```

Listing 1: Clone application

[1] https://wwwmatthes.in.tum.de/pages/g78qcvnwz3u3/Web-Technologies-Frameworks-Libraries-and-Plattforms
[2] https://docs.djangoproject.com/en/1.7/intro/install/

1.2.2 Create

If you want to start from scratch, create your development directory and create the application yourself. The application will be stored within the directory scorecard.

```
django-admin startproject webtech
cd webtech
./manage.py startapp scorecard
./manage.py migrate
```

Listing 2: Create application

Edit the file *webtech/settings.py* and add your newly created django app.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'scorecard',
)
```

Listing 3: Register application

You might need to create some directories and template files yourself if you do not use the template from GitHub. This is currently not supported and most likely will never be.

1.3 Test the application

To start the server run the following command. You can then access the django application in your browser via http://127.0.0.1:8000/scorecard

```
./manage.py runserver
```

Listing 4: Run development server

2 Models

2.1 Course model

Within models information is stored for the web application. We use one model witch stores a course and the number of votes it got. A positive vote increases the vote counter. A negative vote decreases it. Create the file *scorecard/course.py* and fill it with the following content:

```
from django.db import models


class Course(models.Model):
    """
    This class stores the information about one course
    course_title    is the field which stores the title of a course
    vote            is the number of votes a course got
    pk              is created automatically as the primary key
    """
    course_title = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Listing 5: scorecard/models.py

As you can see, there is one character field with maximum length of 200 characters, that stores teh title of a course. Additionally there is a field votes that stores the number of votes one course got. The number of votes can go negative. Now the model has to be registered with our django installation. Run:

```
./manage.py makemigrations scorecard
./manage.py migrate
```

Listing 6: Register models

2.2 Admin app for editing models

If you try to access your django application (http://127.0.0.1:8000/scorecard, you remember?), you will get an error message. We have not yet defined the framework to show anything. To see if your model was propperly created and interact with it, django provides an admin interface in its package django.contrib.admin. This one is activated by default if you create a new django application.

To be shown on the admin interface, the new model needs to be enabled. You can do this by adding the model to the file *scorecard/admin.py*.

```
from django.contrib import admin
from scorecard.models import Course

admin.site.register(Course)
```

Listing 7: scorecard/admin.py

To access it, you need to create a user first (In the version from github, the credentials are admin:admin). Run:

```
./manage.py createsuperuser
```

Listing 8: Create superuser

Now you can access http://127.0.0.1:8000/admin and create some courses as you like. As you can see, the course overview is not named nicely. This is because django does not know how to convert the course objects to a string. Add a `__str__` method to the model to solve this. We let the string conversion return the title of the course, as this is human readable and for our case identifies the courses.

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

```
class Course(models.Model):
    """
    This class stores the information about one course
    course_title    is the field which stores the title of a course
    vote            is the number of votes a course got
    pk              is created automatically as the primary key
    """
    course_title = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    def __str__(self):
        """
        Convert a Course Model to a human readable string
        :return: Returns the title of the course
        """
        return self.course_title
```

Listing 9: scorecard/models.py

The admin application can be highly modified to match your needs concerning ordering or search of model content. However this is not covered in this tutorial. Please refer to the official documentation for more information.

3 Views

If you read further, you might notice that what is called views in django as mostly called controllers. Do not get bothered by that. It is correct. What is usually called view is named templates in django [3]. This is just a naming convention of django and has no effect on our code.

3.1 Base views

To make django return something when accessed using the browser, we need views. Put the following into *scorecard/views.py*. This creates a view for the index, where a static string is returned. For the votes page the view shows a string that contains some variables it got delivered. These views are only here to show a basic concept of views. We will put there some real content later on. The request variable is given to the views and identifies the request. As a return value you will send a HttpResponse.

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


def index(request):
    return HttpResponse("This is the index page.")


def vote(request, pk, vote):
    return HttpResponse("You are voting on %s with %s." % (pk, vote))
```

Listing 10: scorecard/views.py

3.2 URL redirects

We now need to make sure, that the corresponding URLs are given to the appropriate views. Django takes the regular expressions defined in the *urls.py* files and gives those to the appropriate views. We create a file *scorecard/urls.py*, register the URLs in there and register this file in the file *webtech/urls.py*. This is because of the application separation that is part of django. Calls that go to `/scorecard/` are identified by the main url configuration and given to the one for our application. If there is nothing more in the url (identified by `^$`), the index view is called. If there url has an arbitrary number of digits and then a 1 or -1, the call is given to the vote function. The additional URL parts are given to the voting as variables pk and votes. This is identified by the regular expression `^(?P<pk>\d+)/(?P<vote>-?1)/$`.

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

from scorecard import views

urlpatterns = patterns('',
                       url(r'^$', views.index, name='index'),
                       url(r'^(?P<pk>\d+)/(?P<vote>-?1)/$', views.vote, name='
                           vote'),
                       )
```

Listing 11: scorecard/urls.py

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

urlpatterns = patterns('',
                       url(r'^admin/', include(admin.site.urls)),
                       url(r'^scorecard/', include('scorecard.urls', namespace=
                           "scorecard")),
```

[3] https://docs.djangoproject.com/en/1.7/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

)

Listing 12: webtech/urls.py

You can now access pages of the following format:

- http://127.0.0.1:8000/scorecard/
- http://127.0.0.1:8000/scorecard/1/1
- http://127.0.0.1:8000/scorecard/1/-1

You will see, that these views are neither connected to your models nor look nicely. The next step is to create some templates to polish up the view and implement the business logic within the views.

4 Templates

Now let's beautify our websites a bit. We will take care of the functionality later. In the directory *scorecard/templates/scorecard* there is a default template already. We will fill it up with some content. In *scorecard/static/scorecard* there is a place for static files. I have put a css file and some logos there already.

4.1 Index template

In the index template we can refer to template commands such as urls, static files or variables with {% %}. Command structures such as if statements or loops are set in {{ }}. Within the index file, we first check, if there is any object_list given. If not, this means, that there is no course stored in the model and we need to show an error. If there are courses given, we iterate over the courses and show the course name with its votes and links to vote the course up or down. `{{ course }}` relates to the course title, because of the `__str__` function, we defined earlier. With `url` you can build urls based on the names defined in the URL configuration files. `scorecard:vote` means, that the url shall be build from the URL with the name vote within the namespace scorecard. Create the file *scorecard/templates/scorecard/index.html* with the content:

```
<!-- Make the connection to djangos static file system -->
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Scorecard</title>
    <!-- Load stylesheets as static files -->
    <link rel="stylesheet" type="text/css" href="{% static 'scorecard/bootstrap
        /css/bootstrap.css' %}" />
    <link rel="stylesheet" type="text/css" href="{% static 'scorecard/style.css
        ' %}" />
</head>
    <div class="page-header"><h1>Scorecard <small>A <a href="https://wwwmatthes
        .in.tum.de/pages/g78qcvnwz3u3/Web-Technologies-Frameworks-Libraries-and-
        Plattforms" target="_blank">webtech</a> page by <a href="http://
        phynformatik.de/" target="_blank">Janosch Maier</a></small> <span id="
        tumlogo">TUM</span><span id="inlogo">in.tum</span></h1></div>
    <!-- Check if there are any courses -->
    {% if object_list %}
    <ul>
    <!-- Iterate over the courses -->
    {% for course in object_list %}
        <!-- List a course and the voting possibilities -->
        <li><strong>{{ course }}</strong> has {{ course.votes }} vote{{ course.
            votes|pluralize }}
            (<a href="{% url 'scorecard:vote' course.pk 1 %}"><span class="
                glyphicon glyphicon glyphicon-thumbs-up" aria-hidden="true"></
                span></a>
            <a href="{% url 'scorecard:vote' course.pk -1 %}"><span class="
                glyphicon glyphicon glyphicon-thumbs-down" aria-hidden="true"></
                span></a>)
        </li>
    {% endfor %}
    </ul>
    {% else %}
        <!-- Show an error if no course existing -->
        <div class="alert alert-info" role="alert">No courses available</div>
    {% endif %}
</body>
```

```
</html>
```

Listing 13: scorecard/templates/scorecard/index.html

4.2 Updated view

To make use of the template file, we need to update the view. The generic views classes allow the programmer to use predefined classes for recurring patterns such as lists of elements or detail views. We just give the view class the name of its template file and the model to work with. Additionally we order the list of courses by the number of votes.

```
from django.shortcuts import render
from django.http import HttpResponse
from django.views import generic

from scorecard.models import Course

class IndexView(generic.ListView):
    """
    The index page is a generic ListView.
    All Courses are shown in a list
    """
    template_name = 'scorecard/index.html'  # Template to use
    model = Course                          # Model to use

    def get_queryset(self):
        """
        Order the Ccurses by their votes begining from the course with most
            votes.
        """
        return Course.objects.get_queryset().order_by('-votes')

def vote(request, pk, vote):
    return HttpResponse("You are voting on %s with %s." % (pk, vote))
```

Listing 14: views.py

4.3 Updated URLs

The new view class need to be called, when the index page is requested. Therefore we need to change the *scorecard/urls.py* file.

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

from scorecard import views

urlpatterns = patterns('',
                       url(r'^$', views.IndexView.as_view(), name='index'),
                       url(r'^(?P<pk>\d+)/(?P<vote>-?1)/$', views.vote, name='
                           vote'),
                       )
```

Listing 15: scorecard/urls.py

5 Business Logic

What is now still missing is the voting function. We already have defined the votes view in the urls files. If such a link is clicked, we will check its input values and update the votes counter. Afterwards the view redirects the user back to the index file.

```
from django.shortcuts import get_object_or_404
from django.http import HttpResponseRedirect, Http404
from django.views import generic
from django.core.urlresolvers import reverse

from scorecard.models import Course

class IndexView(generic.ListView):
    """
    The index page is a generic ListView.
    All Courses are shown in a list
    """
    template_name = 'scorecard/index.html'  # Template to use
    model = Course                          # Model to use

    def get_queryset(self):
        """
        Order the Ccurses by their votes begining from the course with most
            votes.
        """
        return Course.objects.get_queryset().order_by('-votes')

def vote(request, pk, vote):
    """
    You can either up- or down-vote a course
    The vote is saved in the model
    Then the user is redirected to the index page
    :param request: Request to work on
    :param pk:      Primary key of the Course
    :param vote:    1 or -1 depending on up/down-vote
    :return:        Redirect to Index view or 404 error
    """
    # If there is no course with the corresponding pk return an error
    course = get_object_or_404(Course, pk=pk)
    if(vote == '1'):
        # Increase Vote
        course.votes += 1
        course.save()
        return HttpResponseRedirect(reverse('scorecard:index'))
    elif(vote == '-1'):
        # Decrease Vote
        course.votes -= 1
        course.save()
        return HttpResponseRedirect(reverse('scorecard:index'))
    else:
        # Vote invalid
        return Http404('Vote must be either 1 or -1')
```

Listing 16: scorecard/views.py

Now the voting system is fully functional.

6 Tests

Django has a sophisticated testing framework. You can directly test your functions (unit tests) and do also further tests that simulate requests. To create some basic tests for the scorecard application put the following into *scorecard/tests.py*. Test-driven development and continuous integration are posible with django. The create courses function acts directly on the model and creates some courses. Those are only valid within the test case function that calls it. Otherwise a test database is used and all data stored in the developing environment are not touched by those tests. The client get methods simulaties request on the webpage and the reverse function is used to create the correct URL based on the view to be called and the parameters given. Assertion is done like in regular unit tests.

```
from django.test import TestCase
from django.core.urlresolvers import reverse

from scorecard.models import Course

def create_courses():
    """
    Create some courses
    """
    Course.objects.create(course_title="EADS", votes=0)
    Course.objects.create(course_title="ITSec", votes=0)
    Course.objects.create(course_title="Webtech", votes=0)

class ScoreboardTest(TestCase):

    def test_if_error_msg_if_no_course(self):
        """
        If there is no Course existing, the Index should present an error msg
        """
        response = self.client.get(reverse('scorecard:index'))
        self.assertContains(response, 'No courses available')

    def test_if_courses_are_shown_without_voting(self):
        """
        If courses are created, they should be shown on the index page
        """
        create_courses()
        response = self.client.get(reverse('scorecard:index'))
        self.assertQuerysetEqual(response.context['object_list'], ['<Course:
            EADS>', '<Course: ITSec>', '<Course: Webtech>'])

    def test_if_voting_redirect_is_correct(self):
        """
        If a vote is counted, the user is redirected
        """
        create_courses()
        response = self.client.get(reverse('scorecard:vote', kwargs={'pk':1, '
            vote':1}))
        self.assertEqual(response.status_code, 302)

    def test_if_voting_increased_counter(self):
        """
        If a positive vote is counted, the counter should increase
        """
        create_courses()
        response = self.client.get(reverse('scorecard:index'))
        old_votes = response.context['object_list'].get(pk=1).votes
```

```
        response = self.client.get(reverse('scorecard:vote', kwargs={'pk':1, '
            vote':1}))
        response = self.client.get(reverse('scorecard:index'))
        new_votes = response.context['object_list'].get(pk=1).votes
        self.assertEqual(old_votes + 1, new_votes)

    def test_if_voting_decreased_counter(self):
        """
        If a negative vote is counted, the counter should decrease
        """
        create_courses()
        response = self.client.get(reverse('scorecard:index'))
        old_votes = response.context['object_list'].get(pk=1).votes
        response = self.client.get(reverse('scorecard:vote', kwargs={'pk':1, '
            vote':-1}))
        response = self.client.get(reverse('scorecard:index'))
        new_votes = response.context['object_list'].get(pk=1).votes
        self.assertEqual(old_votes - 1, new_votes)
```

Listing 17: scorecard/tests.py

To run the tests, use the following command:

```
./manage.py tests
```

Listing 18: Run tests

7 Messages

With django you can easily create messages and propagate them in between your pages. The messages application is already installed, if you create a new project. You just need to put some few statements into your *webtech/config.py* to make it work. The second statement is used to make error messages work propperly with bootstrap. The css class therefore has to be set to danger, not error.

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.messages.context_processors.messages',
    'django.contrib.auth.context_processors.auth'
)

from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.ERROR: 'danger'
}
```

Listing 19: webtech/config.py

Update your *views.py* in the following way to add some messages.

```
def updateVote(course, vote):
    if vote == '1':
        course.votes += 1
        course.save()
        return  True
    elif vote == '-1':
        course.votes -= 1
        course.save()
        return True
    return False


def vote(request, pk, vote):
    """
    You can either up- or down-vote a course
    The vote is saved in the model
    Then the user is redirected to the index page
    :param request: Request to work on
    :param pk:      Primary key of the Course
    :param vote:    1 or -1 depending on up/down-vote
    :return:        Redirect to Index view or 404 error
    """
    # If there is no course with the corresponding pk return an error
    course = get_object_or_404(Course, pk=pk)
    if updateVote(course):
        messages.add_message(request, messages.SUCCESS, "Vote Successful")
        return HttpResponseRedirect(reverse('scorecard:index'))
    else:
        # Vote invalid
        messages.add_message(request, messages.WARNING, "Vote Not Successful")
        return HttpResponseRedirect(reverse('scorecard:index'))
```

Listing 20: Add messages to views

To show the messages on your views page, use the following code. This uses djangos message.tags attribute to get the alert class for bootstrap.

```
    <!-- Show message if stored -->
    {% if messages %}
        {% for message in messages %}
            <div class="alert alert-{{ message.tags }}">{{ message }}</div>
```

```
    {% endfor %}
{% endif %}
```

Listing 21: index.html

8 Session

You need to add the request context_processor, so you can access the request.session variable in the view/template

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.messages.context_processors.messages',
    'django.contrib.auth.context_processors.auth',
    'django.core.context_processors.request'
)
```

Listing 22: webtech/config.py

Update your *views.py* in the following way to add the session storage handling.

```
    """
    You can either up- or down-vote a course
    The vote is saved in the model
    Then the user is redirected to the index page
    :param request: Request to work on
    :param pk:      Primary key of the Course
    :param vote:    1 or -1 depending on up/down-vote
    :return:        Redirect to Index view or 404 error
    """
    # If there is no course with the corresponding pk return an error
    course = get_object_or_404(Course, pk=pk)
    if request.session.get('has_voted', False):
        messages.add_message(request, messages.ERROR, "You have already voted!"
            )
        return index()
    if updateVote(course, vote):
        request.session['has_voted'] = True
        messages.add_message(request, messages.SUCCESS, "Vote Successful!")
        return index()
    else:
        # Vote invalid
        messages.add_message(request, messages.ERROR, "Vote Not Successful!")
        return index()

def vote_again(request):
    request.session['has_voted'] = False
    return index()
```

Listing 23: Add messages to views

To show the messages on your views page, use the following code. This uses djangos message.tags attribute to get the alert class for bootstrap.

```
    {% if request.session.has_voted %}
        <div class="alert alert-info">
        You usually can only vote once. If you want to vote again, I can make
            an exception for you. <a href="{% url 'scorecard:vote_again' %}">
            Vote Again?</a>
        </div>
    {% endif %}
```

Listing 24: index.html

Add a new URL redirect so to make everything work. To show the messages on your views page, use the following code. This uses djangos message.tags attribute to get the alert class for bootstrap.

```
urlpatterns = patterns('',
                       url(r'^$', views.IndexView.as_view(), name='index'),
```

```
    url(r'^(?P<pk>\d+)/(?P<vote>-?\d)/$', views.vote, name='
        vote'),
    url(r'^vote_again$', views.vote_again, name='vote_again'
        )
)
```

Listing 25: urls.py

9 Translatuon

Create *scorecard/locale*.
Run from within *scorecard*:

```
django-admin.py makemessages -l de
```

Listing 26: webtech/config.py

Then Edit the .po file
and run:

```
django-admin.py compilemessages
```

Listing 27: Add messages to views

Now create your language strings