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

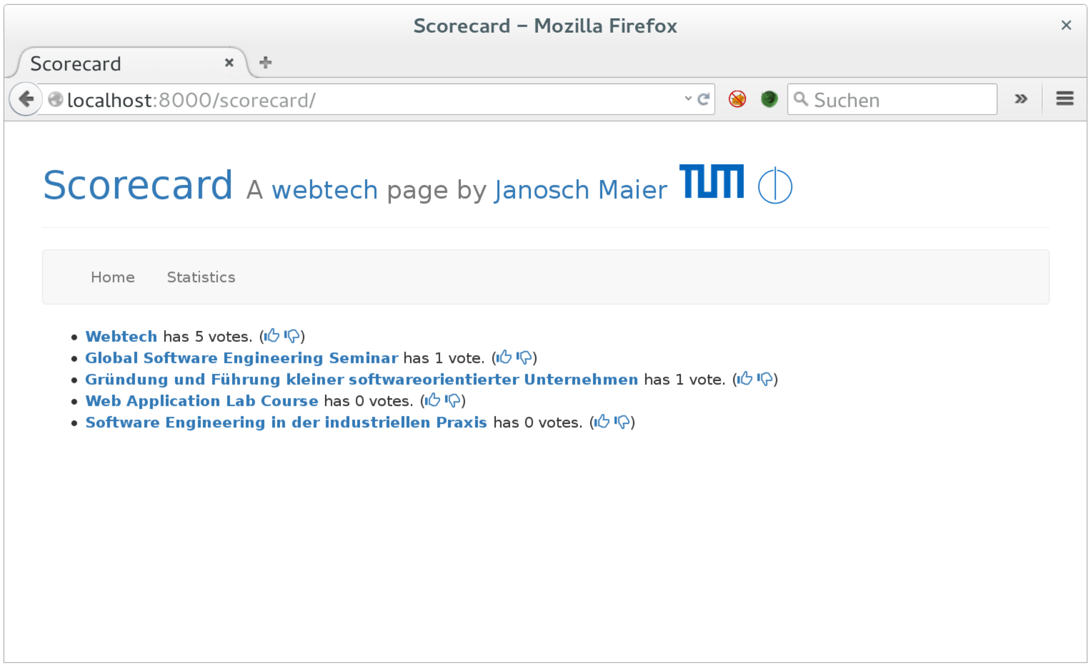
Janosch Maier <maierj@in.tum.de>

April 09, 2015

Contents

1 Introduction

This tutorial introduces advanced django features within a scorecard application. Seminars and lectures can be added and then voted up or down. An overview page shows the courses ranked by their votes. Each course belongs to a certain lecturer. A statistics page shows which lecturer performs best by the votes mean of his courses. This tutorial was build for the TUM seminar Webtech within the summer term 2015 [1]. The application is shown in figure 1.


Scorecard – Mozilla Firefox
Scorecard
localhost:8000/scorecard/
Suchen
Scorecard A webtech page by Janosch Maier
Home
Statistics

- **Webtech** has 5 votes.
- **Global Software Engineering Seminar** has 1 vote.
- **Gründung und Führung kleiner softwareorientierter Unternehmen** has 1 vote.
- **Web Application Lab Course** has 0 votes.
- **Software Engineering in der industriellen Praxis** has 0 votes.

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. The tutorial should work on a Linux system without modification. On Windows you will probably have slightly different commands to run the `manage.py` script.

1.2 Setup

To get the application from the GitHub use:

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

Listing 1: Clone application

You will find several applications in the GitHub directory:

- *webtech* will contain the full webtech application
- *webtech-tranlation* will contain the full webtech application including German translations
- *webtech-skeleton* will contain a skeleton to work with this tutorial yourself

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

To start the server, descend to one of those three directories and 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 2: Run development server

2 Object-Relational Mapping

Django stores the objects of its models in a relational database. To access these from within the programming environment, Object-Relational Mapping (ORM) is used. Each model is represented by a table with the model's attributes as its columns. ORM allows the programmer to access the stored models in an easy way.

2.1 Lecturer and Course Model

The following models are used for the scoreboard application:

```python
from django.db import models


class Lecturer(models.Model):
    """
    This class stores the information about one lecturer
    first_name   is the field which stores the first name
    last_name    is the field which stores the last name
    pk           is created automatically as the primary key
    """
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)

    def __str__(self):
        """
        Convert the Lecturer Name to a human readable string
        :return: Returns the full name of the lecturer
        """
        return "{0} {1}".format(self.first_name, self.last_name)


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
    lecturer         refers to the Lecturer model and identifies the lecturer
       responsible for the course
    pk               is created automatically as the primary key
    """
    course_title = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
    lecturer = models.ForeignKey(Lecturer)

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

    def is_great_course(self):
        """
        Figures out if a course is great or not.
        Great courses have more than 100 positive votes
        :return: True if votes > 100, Fales otherwise
        """
        return self.votes > 100
```

Listing 3: scorecard/models.py

2.2 Adding Objects

To work with the models, objects need to be created. The command `Lecturer()` respectively `Course()` create a new object of the corresponding class. The `save()` method stores the object permanently in the database.

```python
from scorecard.models import Course, Lecturer

l = Lecturer(first_name="Alexander", last_name="Waldmann")
l.save()
c = Course(course_title="Webtech", votes=0, lecturer=l)
c.save()
```

Listing 4: Adding Objects

Those (and the following) commands can be either run within your django project classes (perhaps the views or a test class as described in the next chapter) or from the interactive shell. To enter the shell run from the terminal:

```
./manage.py shell
```

Listing 5: Open django shell

2.3 Retrieving Objects

The attribute `pk` is the primary key of an object. If not specified otherwise, it is an integer, that is automatically incremented. To get the objects we just created, you can call:

```python
l = Lecturer.objects.get(pk=1)
c = Course.objects.get(pk=1)
```

Listing 6: Retrieving objects by Primary Key

To get all objects of one model, you can use the `all()` method. This may be used for iterating or showing a set of objects in a template. You should probably add some more objects in order to get meaningfull results.

```python
for course in Course.objects.all():
  print(course)
```

Listing 7: Retrieving all objects of a model

To order the objects shown, there is the `order_by()` method available. Order by can take several attributes for ordering. By default the order is ascending. A negative sign in front of the attribute name means descending ordering. The following statements gets all Lecturers ordered first by their first name, then by their last name in descending ordering. The following commands are similar:

```python
Lecturer.objects.all().order_by('-first_name', '-last_name')
Lecturer.objects.order_by('-first_name', '-last_name')
```

Listing 8: Retrieving objects in order

To restrict the output of objects to a certain amount or range, one can access the output of an objects method like an array. To get the three courses with the most votes, run:

```python
Course.objects.order_by('-votes')[0:3]
```

Listing 9: Retrieving ranges of objects

2.4 Filtering

Most likely, you will not want to get all objects, that are stored but filter your objects by certain criteria. For filtering one can use the `filter()` method. To filter an object by its attributes, suffixes like `__contains`, `__startswith`, `__exact`, `__gte` are used. The filter in line 1 will match the Webtech course, that was created at the beginning of this chapter. The filter in line 2 will not match the course (at least if you use a database that has case-sensitive comparisons; So when using sqlite as database backend it will match the course as well). The filter in line 3 will match again, as the letter `i` means, that the comparison is case-insensitive. The filter in line 4 matches all courses whose vote counter is greater than 3. The method `exclude()` works the same way as `filter()`, with all but the matched objects returned.

```
Course.objects.filter(course_title__exact='Webtech')        1
Course.objects.filter(course_title__contains='web')         2
Course.objects.filter(course_title__icontains='web')        3
Course.objects.filter(votes__gte=3)                         4
```

Listing 10: Filtering objects

Filter who take another attribute of the same object into account use the `F()` method. To find all lecturers whose first name is the same as their last name, use:

```
from django.db.models import F                                    1
Lecturer.objects.filter(first_name__exact=F('last_name'))         2
```

Listing 11: Filtering objects with attribute comparisons

Several filter statements are connected with a logical and. To create more sophisticated queries for example with or statements or negations, one can use the `Q()` method.

The filter statements in the 2nd and 3rd line yield the same results. They return all lecturers whose first name contains the string Alex *and* the last name does not contain the string Shumaiev.

The statement in the 4th line however returns all lecturers whose first name contains Alex *or* the last name contains Shumaiev.

The last statement filters all lecturers whose first name contains Alex *or* the last name contains *not* Shumaiev.

```
from django.db.models import Q                                                   1
Lecturer.objects.filter(first_name__contains='Alex', last_name__contains='       2
    Shumaiev')
Lecturer.objects.filter(Q(first_name__contains='Alex'), Q(last_name__contains='  3
    Shumaiev'))
Lecturer.objects.filter(Q(first_name__contains='Alex') | Q(last_name__contains=  4
    'Shumaiev'))
Lecturer.objects.filter(Q(first_name__contains='Alex') | ~Q(last_name__contains  5
    ='Shumaiev'))
```

Listing 12: Filtering objects with or statements and negations

2.5 Aggregation

With a relational database backend it is not only possible to filter out elements but do calculations already in the database. The following command counts the number of courses that are stored:

```
Course.objects.count()        1
```

Listing 13: Counting objects

Further aggregation needs the corresponding imports from the django model class.

The command in line 2 and 3 get the average, respectively maximum votes of the stored courses.

The filter in line 4 gives all courses, who are voted highest of all courses.

```
from django.db.models import Avg, Max                                            1
Course.objects.aggregate(Avg('votes'))['votes__avg']                             2
Course.objects.aggregate(Max('votes'))['votes__max']                             3
Course.objects.filter(Q(votes__exact=Course.objects.aggregate(Max('votes'))['    4
    votes__max']))
```

Listing 14: Average and Maximum values of objects

2.6 Annotation

Temporary attributes that depend other objects are helpful in some cases. To find out which lecturer performs best, one can group the courses by lecturers and add the average votes of all his courses.

```
Course.objects.values('lecturer').annotate(avg_votes=Avg('votes'))
```

Listing 15: Annotating objects

2.7 Raw SQL Queries

Using the `raw()` method it is also possible to write raw SQL statements. This gives the programmer much freedom, however unescaped SQL statements may open the application for SQL injection attacks. So use this feature wisely. I will only refer to the django manual at this point[3].

2.8 Examples

Several of those ORM techniques are used to create the statistics page for the scoreboard application. The `get_best_lecturer_with_mean()` function returns the lecturer object with the highest votes mean and the votes mean itself. The statistics view sets the context for the information to be shown in the template.

```
def get_best_lecturer_with_mean():
    avg_votes = Course.objects.values('lecturer').annotate(avg_votes=Avg('votes'))
    max_avg = avg_votes.aggregate(max_avg=Max('avg_votes'))['max_avg']
    best_lecturer_query_result = avg_votes.filter(avg_votes__exact=max_avg
    best_lecturer = Lecturer.objects.get(pk=best_lecturer_query_result[0]['lecturer'])
    return best_lecturer, max_avg

def statistics(request):
    """
    Show statistics page
    :param request:
    :return:
    """
    best_lecturer, best_lecturer_mean = get_best_lecturer_with_mean()
    context = {
        'lecturer_count': Lecturer.objects.count(),
        'courses_count': Course.objects.count(),
        'courses_votes_mean': Course.objects.aggregate(Avg('votes'))['votes__avg'],
        'lecturer_best': best_lecturer,
        'lecturer_best_votes_mean': best_lecturer_mean,
    }
    return render(request, 'scorecard/statistics.html', context)
```

Listing 16: Statistics view

[3]https://docs.djangoproject.com/en/1.7/topics/db/sql/

3 Tests

Django has a sophisticated testing framework. You can directly test your functions with unit tests and do also further tests that simulate requests. The file *scorecard/tests.py* contains the tests for the scorecard application. Test-driven development and continuous integration are posible with django.

3.1 Configure the Testing Environment

The class ScoreboardTest contains the tests for the scoreboard application. When tests are run, the application creates an extra database for the tests. all data in your productive environment is not touched. The `setUp()` method (line 14) creates objects for the test cases. It is run automatically by the testing environment before each test. The test methods will use the defined lecturers and courses to run their tests.

```
1  from django.test import TestCase
2  from django.core.urlresolvers import reverse
3
4  from scorecard.models import Course, Lecturer
5
6
7  class ScoreboardTest(TestCase):
8      lecturer_1 = Lecturer()
9      lecturer_2 = Lecturer()
10     course_1 = Course()
11     course_2 = Course()
12     course_3 = Course()
13
14     def setUp(self):
15         """
16         Create some courses & lecturers
17         """
18         self.lecturer_1 = Lecturer.objects.create(first_name="Janosch",
               last_name="Maier")
19         self.lecturer_2 = Lecturer.objects.create(first_name="A", last_name="B"
               )
20         self.course_1 = Course.objects.create(course_title="EADS", votes=0,
               lecturer=self.lecturer_1)
21         self.course_2 = Course.objects.create(course_title="ITSec", votes=0,
               lecturer=self.lecturer_2)
22         self.course_3 = Course.objects.create(course_title="Webtech", votes=0,
               lecturer=self.lecturer_2)
```

Listing 17: Exceprt from scorecard/tests.py

The test functions use several other methods for convenience. Important is `vote()` (line 10), which takes a course and either 1 or -1 and votes the course up and down. The function returns the votes before and after the vote. This is not done directly within the model but using the views. `self.client.get()` runs a get request on a page of the project. `reverse` does a lookup in the *urls.py* file to get the correct URL. `vote_again()` (line 24) calls the vote_again page to reenable the voting if a user in this session has already voted. This functionality is covered in section 4.2 of this tutorial.

```
1      def delete_courses(self):
2          """
3          Delete all courses for test
4          :return:
5          """
6          self.course_1.delete()
7          self.course_2.delete()
8          self.course_3.delete()
9
10     def vote(self, course, up_down):
        """
        Vote for course with vote up_down
        :param course: Course object
        :param up_down: 1 or -1
        :return: old_votes, new_votes
        """
        response = self.client.get(reverse('scorecard:index'))
        old_votes = response.context['object_list'].get(pk=course.pk).votes
        self.client.get(reverse('scorecard:vote', kwargs={'pk': course.pk, '
            vote': up_down}))
        response = self.client.get(reverse('scorecard:index'))
        new_votes = response.context['object_list'].get(pk=course.pk).votes
        return old_votes, new_votes

    def vote_again(self):
        """
        Reset has_already_voted hook
        :return:
        """
        self.client.get(reverse('scorecard:vote_again'))
```

Listing 18: Exceprt from scorecard/tests.py

3.2 Unittests

test_lecturer_str() works directly on the model and checks if the __str__() method of the lecturer model works correctly

```
    def test_lecturer_str(self):
        """
        Test if the __str__ method of lecturer works properly
        :return:
        """
        lecturer = Lecturer.objects.create(first_name='Janosch', last_name='
            Maier')
        self.assertEqual(lecturer.__str__(), 'Janosch Maier')
```

Listing 19: Exceprt from scorecard/tests.py

test_great_course_for_bad_course() and test_great_course_for_great_course() also work on the model directly. They use the function is_great_course(), which is defined in the course model. Those are regular unit tests.

```
    def test_great_course_for_bad_course(self):
        """
        Return false if course has less then 100 votes
        :return:
        """
        course = Course.objects.create(course_title="EADS", votes=100, lecturer
            =self.lecturer_1)
        self.assertEqual(course.is_great_course(), False)

    def test_great_course_for_great_course(self):
        """
        Return false if course has less then 100 votes
        :return:
        """
        course = Course.objects.create(course_title="EADS", votes=101, lecturer
            =self.lecturer_1)
        self.assertEqual(course.is_great_course(), True)
```

Listing 20: Exceprt from scorecard/tests.py

3.3 Testing Webserver Respnoses

`test_error_msg_if_no_course()` first deletes all courses and then checks if the response for a get request on the index page contains the error message, that there are no courses available. This check is done using the `assertContains()` method.

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

Listing 21: Exceprt from scorecard/tests.py

`test_if_courses_are_shown_without_voting()` checks if all courses created in the `setUp()` method are given to the index template correctly. `assertQuerysetEqual()` checks if the object_list in the context equals the one that contains all the defined courses for the test case.

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

Listing 22: Exceprt from scorecard/tests.py

`test_if_voting_redirect_is_correct()` looks if the status code of the response when the voting page is called is a correct redirect.

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

Listing 23: Exceprt from scorecard/tests.py

The following 4 test methods test the voting functionality. A vote shall only be counted when the user in the session has not yet voted (again, see section 4.2 for more details). They use the `vote()` method to call the voting page and compare the votes before and afterwards.

```
def test_if_voting_increased_counter(self):
    """
    If a positive vote is counted, the counter should increase
    """
    vote = 1
    old_votes, new_votes = self.vote(self.course_1, vote)
    self.assertEqual(old_votes + vote, new_votes)

def test_if_voting_decreased_counter(self):
    """
    If a negative vote is counted, the counter should decrease
    """
    vote = -1
    old_votes, new_votes = self.vote(self.course_1, vote)
    self.assertEqual(old_votes + vote, new_votes)

def test_if_voting_increased_counter_twice_without_reset(self):
    """
    If a positive vote is counted, the counter should increase
    """
    vote = 1
    old_votes, not_used = self.vote(self.course_1, vote)
    not_used, new_votes = self.vote(self.course_1, vote)
    self.assertEqual(old_votes + vote, new_votes)

def test_if_voting_increased_counter_twice_with_reset(self):
    """
    If a positive vote is counted, the counter should increase
    """
    vote = 1
    old_votes, not_used = self.vote(self.course_1, vote)
    self.vote_again()
    not_used, new_votes = self.vote(self.course_1, vote)
    self.assertEqual(old_votes + (2 * vote), new_votes)
```

Listing 24: Exceprt from scorecard/tests.py

`test_statistic()` checks if the statistic pages contains the correct values that should be calculated with the lecturer and course objects as created using `setUp()`. lecturer_1 has only one course that is voted with two positive votes. The other lecture has an average of zero votes. Therefore lecturer_1 is the best lecturer with a votes mean of two.

```
    def test_statistic(self):
        """
        Statistic should show correct values
        :return:
        """
        vote = 1
        self.vote(self.course_1, vote)
        self.vote_again()
        self.vote(self.course_1, vote)
        response = self.client.get(reverse('scorecard:statistics'))
        self.assertEqual(response.context['lecturer_best'], self.lecturer_1)
        self.assertEqual(response.context['lecturer_best_votes_mean'], 2)
```

Listing 25: Exceprt from scorecard/tests.py

To run the tests, use the following command:

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

Listing 26: Run tests

4 Middleware

Middlewares hook in between the request and the view. This means you can set additional information in the view easily or retrieve information that was not explicitly provided by the user with the request. Two prominent examples of this are the sessions and the messages module of django.

4.1 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.

4.1.1 Configure Messages

You just need to put some few statements into your *webtech/config.py* to make it work. The context processor messages (line 2) makes your messages available for the templates. The second statement (line 6–9) is used to make error messages work propperly with bootstrap. The css class therefore has to be set to danger, not error.

```
1  TEMPLATE_CONTEXT_PROCESSORS = (
2      'django.contrib.messages.context_processors.messages',
3      'django.contrib.auth.context_processors.auth'
4  )
5
6  from django.contrib.messages import constants as messages
7  MESSAGE_TAGS = {
8      messages.ERROR: 'danger'
9  }
```

Listing 27: webtech/config.py

4.1.2 Adding Messages to Views

The `message.add_message()` method in the *views.py* file adds some messages that are dilivered to the templates.

```
from django.contrib import messages

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 28: Add messages to views

4.1.3 Showing Messages in Templates

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 messages %}
    {% for message in messages %}
        <div class="alert alert-{{ message.tags }}">{{ message }}</div>
    {% endfor %}
{% endif %}
```

Listing 29: index.html

In general, one could also create a message variable in the view context himself. But the messaging framework simplifies this task and provides easy functions for this purpose.

4.2 Session

With the session middleware, one can store additional information for the requests that are connected with a session. In this scoreboard each user (identified by its session) shall only vote once. Therefore we store a hook in the session, if the user has already voted. For testing purposes, we allow the user to reset the hook by calling a special page.

4.2.1 Configure Sessions

You need to add the request context_processor in *scoreboard/config.py*, 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 30: webtech/config.py

4.2.2 Adding Sessions to Views

Update your *views.py* in the following way to add the session storage handling. With `request.session.get('has_voted', False)` (line 5) we try to retrieve the value of the has_voted session variable. If there is nothing stored, the function returns `False`. To store True in the variable you can simply use `request.session['has_voted'] = True` (line 2).

```
 1  def set_voted(request, voted):
 2      request.session['has_voted'] = voted
 3
 4  def has_already_voted(request):
 5      return request.session.get('has_voted', False)
 6
 7  def vote(request, pk, vote):
 8      """
 9      You can either up- or down-vote a course
10      The vote is saved in the model
11      Then the user is redirected to the index page
12      :param request: Request to work on
13      :param pk:      Primary key of the Course
14      :param vote:    1 or -1 depending on up/down-vote
15      :return:        Redirect to Index view or 404 error
16      """
17      # If there is no course with the corresponding pk return an error
18      course = get_object_or_404(Course, pk=pk)
```

```
19        if has_already_voted(request):
20            messages.add_message(request, messages.ERROR, "You have already voted!"
                 )
21        elif update_vote(course, vote):
22            set_voted(request, True)
23            messages.add_message(request, messages.SUCCESS, "Vote Successful!")
24        else:
25            # Vote invalid
26            messages.add_message(request, messages.ERROR, "Vote Not Successful!")
27        return redirect_to_index()
28
29    def vote_again(request):
30        """
31        Remove vote counter from session
32        :param request:
33        :return:
34        """
35        set_voted(request, False)
36        return redirect_to_index()
```

Listing 31: Session handling in vote view

4.2.3 Using Sessions in Templates

As we want the user to reset its session, so he can vote again, we show in information message if the has_voted variable is set. The link should bring the user to the vote_again page that resets the session hook (line 25 in listing 31)

```
1    {% if request.session.has_voted %}
2        <div class="alert alert-info">
3        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>
4        </div>
5    {% endif %}
```

Listing 32: index.html

To access make the vote_again page work, add a new URL redirect (line 4).

```
1    urlpatterns = patterns('',
2                           url(r'^$', views.IndexView.as_view(), name='index'),
3                           url(r'^(?P<pk>\d+)/(?P<vote>-?\d)/$', views.vote, name='
                                vote'),
4                           url(r'^vote_again$', views.vote_again, name='vote_again'
                                )
5                           )
```

Listing 33: urls.py

5 Translation

To make your django application available for a broad audience, internationalization is needed. With the translation framework of django this is possible.

5.1 Translation definitions

To create an internationalized version of the scorecard application, all strings need to put into translation functions. Based on these translation strings, the *django-admin.py* file creates a *.po* file, in which the translation takes place. First in your views (or wherever strings are created), import the translation method and enclose the strings accordingly. Look at the string in line 6.

```
1 from django.utils.translation import ugettext as _
2
3 def vote(request, pk, vote):
4     course = get_object_or_404(Course, pk=pk)
5     if has_already_voted(request):
6         messages.add_message(request, messages.ERROR, _("You have already voted!"))
```

Listing 34: exceprt of views.py with translation

For translating templates, the `{% trans "string" %}` tag is available. See how translation for the menu links is initialized in lines 5 and 6 of the following snippet.

```
1 {% load i18n %}
2
3 <div class="navbar-collapse" id="bs-example-navbar-collapse-1">
4     <ul class="nav navbar-nav">
5         <li><a href="{% url 'scorecard:index' %}">{% trans "Home" %}</a></li>
6         <li><a href="{% url 'scorecard:statistics' %}">{% trans "Statistics" %}</a></li>
7     </ul>
8 </div><!-- /.navbar-collapse -->
```

Listing 35: exceprt of base.html with translation

5.2 Pluralization

Pluralization is a bit more difficult. If English is set as the first language, the plural "s" can be used easily with the `pluralize` command. This is not possible for more complicated languages such as German or any than the first language used in the project. To create a pluralized version of a string, you need a counter that defines whether the singular or plural version of the string is used. In the following code, `course.votes` is used as counter, to pluralize the votes.

```
{% blocktrans count counter=course.votes %}{{ counter }} vote.{% plural %}{{ counter }} votes.{% endblocktrans %}
```

Listing 36: exceprt of index.html with translation

5.3 Translation Files

Now we have defined the strings that shall be translated. To do the actual translation, let django create a translation file. Create the folder *scorecard/locale*.

From within the *scorecard* directory run the following command to create a translation file for German:

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

Listing 37: Create a German translation file

Now in the directory *scorecard/locale*, a directory for the German language appears. Deep down in this directory there is a *django.po* file. Create the translation strings within this file. You can see the plural version of the string for the votes counter in the *index.html* file in line 9-14.

```
#: templates/scorecard/base.html:20                1
msgid "Home"                                       2
msgstr "Start"                                     3
                                                   4
#: templates/scorecard/base.html:21                5
msgid "Statistics"                                 6
msgstr "Statistik"                                 7
                                                   8
#: templates/scorecard/index.html:10               9
#, python-format                                   10
msgid "%(counter)s vote."                          11
msgid_plural "%(counter)s votes."                  12
msgstr[0] "%(counter)s Stimme."                    13
msgstr[1] "%(counter)s Stimmen."                   14
```

Listing 38: django.po file for German

To apply the translated messages, run:

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

Listing 39: Compile messages