Coverage for **django/learning_base/views.py** : 74%

456 statements | 339 run | 117 missing | 0 excluded

```
"""
This module contains all directly accessed API functions
Views are not documented extensively in the code but at
https://github.com/Iliricon/clonecademy
"""
from django.http import HttpResponse
from django.core.mail import send_mail
from django.contrib.auth.models import User, Group
from django.utils import timezone
from django.utils.crypto import get_random_string

from rest_framework import status
from rest_framework import authentication, permissions
from rest_framework.views import APIView
from rest_framework.exceptions import ParseError, PermissionDenied
from rest_framework.response import Response

from . import custom_permissions
from . import serializers
from .models import Course, CourseCategory, Try, Profile, started_courses


class CategoryView(APIView):
    """
    Shows, creates, updates and deletes a category
    :author: Claas Voelcker, Tobias Huber
    """
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (custom_permissions.IsAdminOrReadOnly,)

    def get(self, request, format=None):
        """
        Shows the categories
        :author: Claas Voelcker
        :return: a list of all categories
        """
        categories = CourseCategory.objects.all()
        data = serializers.CourseCategorySerializer(categories, many=True).data
        return Response(data,
                        status=status.HTTP_200_OK)

    def post(self, request, format=None):
        """
        everything else but displaying
        :author: Tobias Huber
        """
        data = request.data
        # check if instance shall be deleted
        if 'delete' in data and data['delete'] == 'true':
            if 'id' in data:
                instance = CourseCategory.objects.get(id=data['id'])
                instance.delete()
                return Response(status=status.HTTP_204_NO_CONTENT)
            return Response({'ans': 'a category with the given id'
                                    + ' does not exist'},
                            status=status.HTTP_404_NOT_FOUND)

        # check if an id is given, signaling to update the corresponding cat.
        if 'id' in data:
```

```
60              category_id = data['id']
61              if CourseCategory.objects.filter(id=category_id).exists():
62                  category = CourseCategory.objects.get(id=category_id)
63                  serializer = serializers.CourseCategorySerializer(
64                      category, data=data, partial=True, )
65              else:
66                  return Response(
67                      {'ans': 'a category with the id ' + str(category_id)
68                              + ' does not exist'},
69                      status=status.HTTP_404_NOT_FOUND)
70          else:
71              # else just create a plain serializer
72              serializer = serializers.CourseCategorySerializer(data=data)
73          if serializer.is_valid():
74              serializer.save()
75              return Response(serializer.data,
76                              status=status.HTTP_200_OK)
77          return Response(serializer.errors,
78                          status=status.HTTP_400_BAD_REQUEST)
79
80
81  class MultiCourseView(APIView):
82      """
83      View to see all courses of a language. The post method provides a general
84      interface with three filter settings.
85      @author Claas Voelcker
86      """
87      authentication_classes = (authentication.TokenAuthentication,)
88      permission_classes = (permissions.IsAuthenticated,)
89
90      def get(self, request, format=None):
91          """
92          Not implemented
93          """
94          return Response({'ans': 'Method not allowed'},
95                          status=status.HTTP_405_METHOD_NOT_ALLOWED)
96
97      def post(self, request, format=None):
98          """
99          Returns a set of courses detailed by the query. It expects a request
100         with the keys 'language', 'category', 'type'. The returning JSON
101         corresponds to the values. All values can be empty strings, resulting
102         in all courses being returned.
103         """
104         try:
105             types = ['mod', 'started']
106             categories = [str(x) for x in CourseCategory.objects.all()]
107             languages = [x[0] for x in Course.LANGUAGES]
108             data = request.data
109             r_type = data['type']
110             r_category = data['category']
111             r_lan = data['language']
112
113             # checks whether the query only contains acceptable keys
114             if not ((r_type in types or not r_type)
115                     and (r_category in categories or not r_category)
116                     and (r_lan in languages or not r_lan)):
117                 return Response({'ans': 'Query not possible'},
118                                 status=status.HTTP_400_BAD_REQUEST)
119
120             courses = Course.objects.all()
121             courses = courses.filter(language=r_lan)
122
123             # filter invisible courses if neccessary
124             if not (request.user.profile.is_mod()
125                     or request.user.profile.is_admin()):
```

```
126                 courses = courses.filter(is_visible=True)
127
128             if r_category != '':
129                 category = CourseCategory.objects.filter(
130                     name=r_category).first()
131                 courses = courses.filter(category=category)
132             if r_type == 'mod':
133                 courses = courses.filter(responsible_mod=request.user)
134             elif r_type == 'started':
135                 courses = started_courses(request.user)
136             data = serializers.CourseSerializer(courses, many=True, context={
137                 'request': request}).data
138             return Response(data, status=status.HTTP_200_OK)
139         except Exception as errors:
140             return Response({'ans': 'Query not possible' + str(errors)},
141                             status=status.HTTP_400_BAD_REQUEST)
142
143
144 class CourseEditView(APIView):
145     """
146     contains all the code related to edit a courses
147     TODO: this is probably redundant code
148     @author Leonhard Wiedmann
149     """
150     authentication_classes = (authentication.TokenAuthentication,)
151     permission_classes = (custom_permissions.IsModOrAdmin,)
152
153     def get(self, request, course_id=None, format=None):
154         """
155         Returns all the information about a course with the answers and the
156         solutions
157         """
158         if not course_id:
159             return Response({'ans': 'Method not allowed'},
160                             status=status.HTTP_405_METHOD_NOT_ALLOWED)
161
162         try:
163             course = Course.objects.filter(id=course_id).first()
164             course_serializer = serializers.CourseEditSerializer(
165                 course,
166                 context={
167                     'request': request})
168             data = course_serializer.data
169             return Response(data)
170
171         except Exception as errors:
172             return Response({'ans': str(errors)},
173                             status=status.HTTP_404_NOT_FOUND)
174
175     def post(self, request, course_id=None, format=None):
176         """
177         Not implemented
178         """
179         return Response({'ans': 'Method not allowed'},
180                         status=status.HTTP_405_METHOD_NOT_ALLOWED)
181
182
183 class CourseView(APIView):
184     """
185     Contains all code related to viewing and saving courses.
186     :author: Claas Voelcker
187     """
188     authentication_classes = (authentication.TokenAuthentication,)
189     permission_classes = (
190         custom_permissions.IsModOrAdminOrReadOnly,)
191
```

```
192     def get(self, request, course_id=None, format=None):
193         """
194         Returns a course if the course_id exists. The course, it's
195         modules and questions are serialized.
196 
197         :author: Claas Voelcker
198         :param request: request object containing auth token and user id
199         :param course_id: the id of the required course
200         :param format: unused (inherited)
201         :return: a response containing the course serialization
202         """
203 
204         # if no course id is given, the method was called wrong
205         if not course_id:
206             return Response({'ans': 'Method not allowed'},
207                             status=status.HTTP_405_METHOD_NOT_ALLOWED)
208         try:
209             # fetch the course object, serialize it and return
210             # the serialization
211             course = Course.objects.get(id=course_id)
212             course_serializer = serializers.CourseSerializer(course, context={
213                 'request': request})
214             return Response(course_serializer.data,
215                             status=status.HTTP_200_OK)
216         # in case of an exception, throw a "Course not found" error for the
217         # frontend, packaged in a valid response with an error status code
218         except Exception:
219             return Response({'ans': 'Course not found'},
220                             status=status.HTTP_404_NOT_FOUND)
221 
222     def post(self, request, course_id=None, format=None):
223         """
224         Saves a course to the database. If the course id is provided,
225         the method updates and existing course, otherwise, a new course
226         is created.
227 
228         :author: Tobias Huber, Claas Voelcker
229         :param request: request containig the user and auth token
230         :param course_id: optional: the course id
231                     (if a course is edited instead of created)
232         :param format: unused (inherited)
233         :return: a status response giving feedback about errors or a sucessful
234                     database access to the frontend
235         """
236 
237         data = request.data
238 
239         # checks whether the request contains any data
240         if data is None:
241             return Response({'error': 'Request does not contain data'},
242                             status=status.HTTP_400_BAD_REQUEST)
243 
244         course_id = data.get('id')
245         # Checks whether the name of the new course is unique
246         if (course_id is None) and Course.objects.filter(
247                 name=data['name']).exists():
248             return Response({'error': 'Course with that name exists'},
249                             status=status.HTTP_409_CONFLICT)
250 
251         # adds the user of the request to the data
252         if course_id is None:
253             data['responsible_mod'] = request.user
254         # if the course is edited, check for editing permission
255         else:
256             responsible_mod = Course.objects.get(id=course_id).responsible_mod
257             # decline access if user is neither admin nor the responsible mod
```

```
258             if (request.user.profile.is_admin()
259                     or request.user == responsible_mod):
260                 data['responsible_mod'] = responsible_mod
261             else:
262                 raise PermissionDenied(detail="You're not allowed to edit this"
263                                        + "course, since you're not the"
264                                        + 'responsible mod',
265                                        code=None)
266 
267         # serialize the course
268         course_serializer = serializers.CourseSerializer(data=data)
269 
270         # check for serialization errors
271         if not course_serializer.is_valid():
272             return Response({'error': course_serializer.errors},
273                             status=status.HTTP_400_BAD_REQUEST)
274 
275         # send the data to the frontend
276         else:
277             try:
278                 course_serializer.create(data)
279                 return Response({'success': 'Course saved'},
280                                 status=status.HTTP_201_CREATED)
281             except ParseError as error:
282                 return Response({'error': str(error)},
283                                 status=status.HTTP_400_BAD_REQUEST)
284 
285 
286 class ToggleCourseVisibilityView(APIView):
287     """
288     changes the visibility of a course
289 
290     alternatively sets the visibility to the provided state
291     {
292         "is_visible": (optional) True|False
293     }
294 
295     @author Tobias Huber
296     """
297 
298     authentication_classes = (authentication.TokenAuthentication,)
299     permission_classes = (custom_permissions.IsAdmin,)
300 
301     def post(self, request, course_id):
302         """
303         Sets the course visibility to the given value
304         :param request: request object from the rest dispatcher
305         :param course_id: the id of the course whos visibility shall be changed
306         :return: a REST Response with a meaningfull JSON formatted message
307         """
308         if course_id is None:
309             return Response({'ans': 'course_id must be provided'},
310                             status=status.HTTP_400_BAD_REQUEST)
311         elif not Course.objects.filter(id=course_id).exists():
312             return Response({'ans': 'course not found. id: ' + course_id},
313                             status=status.HTTP_404_NOT_FOUND)
314         else:
315             course = Course.objects.get(id=course_id)
316             if 'is_visible' in request.data:
317                 if not (request.data['is_visible'] == 'true'
318                         or request.data['is_visible'] == 'false'):
319                     Response({'ans':'is_visible must be "true" or "false" of type string'}
320                 course.is_visible = request.data['is_visible'] == 'true'
321             else:
322                 course.is_visible = not course.is_visible
323             course.save()
```

```
            return Response({'is_visible': course.is_visible},
                            status=status.HTTP_200_OK)


class ModuleView(APIView):
    """
    Shows a module
    @author Claas Voelcker
    """
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (permissions.IsAuthenticated,)

    def get(self, request, course_id, module_id, format=None):
        """
        Not implemented
        """
        return Response({'ans': 'Method not allowed'},
                        status=status.HTTP_405_METHOD_NOT_ALLOWED)

    def post(self, request, format=None):
        """
        Not implemented
        """
        return Response({'ans': 'Method not allowed'},
                        status=status.HTTP_405_METHOD_NOT_ALLOWED)


class QuestionView(APIView):
    """
    View to show questions and to evaluate them. This does not return the
    answers, which are given by a separate class.
    @author Claas Voelcker
    """
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (permissions.IsAuthenticated,)

    @staticmethod
    def can_access_question(user, question, module_id, question_id):
        """
        Checks if the question is accessable by the user (all questions before
        need to be answered correctly)
        :param user: user wanting to access
        :param question: question to be accessed
        :param module_id: module id the question belongs to
        :param question_id: the questions id
        :return: True|False (see description)
        @author Tobias Huber
        """
        module = question.module
        first_question = int(module_id) <= 0 and int(question_id) <= 0
        if first_question:
            return True
        elif (not first_question
              and question.get_previous_in_order()
              and Try.objects.filter(
                  user=user,
                  question=question.get_previous_in_order(),
                  solved=True)):
            return True
        elif (not module.is_first_module()
              and module.get_previous_in_order()
              and Try.objects.filter(
                  user=user,
                  question=module.get_previous_in_order().question_set.all()[0],
                  solved=True)):
            return True
```

```
390         return False
391
392     def get(self, request, course_id, module_id, question_id, format=None):
393         """
394         Get a question together with additional information about the module
395         and position (last_module and last_question keys)
396         """
397         try:
398             course = Course.objects.get(id=course_id)
399             course_module = course.module_set.all()[int(module_id)]
400             question = course_module.question_set.all()[int(question_id)]
401
402             if question is None:
403                 return Response({'ans': 'Question not found'},
404                                 status=status.HTTP_404_NOT_FOUND)
405             if not self.can_access_question(request.user, question, module_id,
406                                             question_id):
407                 return Response({'ans': "Previous question(s) haven't been "
408                                         'answered correctly yet'},
409                                 status=status.HTTP_403_FORBIDDEN)
410             data = serializers.QuestionSerializer(question,
411                                                   context={'request': request})
412             data = data.data
413             return Response(data, status=status.HTTP_200_OK)
414         except Exception as error:
415             return Response({'error': str(error)},
416                             status=status.HTTP_404_NOT_FOUND)
417
418     def post(self, request, course_id, module_id, question_id, format=None):
419         """
420         Evaluates the answer to a question.
421         @author Tobias Huber
422         """
423         try:
424             course = Course.objects.get(id=course_id)
425             course_module = course.module_set.all()[int(module_id)]
426             question = course_module.question_set.all()[int(question_id)]
427         except Exception:
428             return Response({'ans': 'Question not found'},
429                             status=status.HTTP_404_NOT_FOUND)
430         # deny access if there is a/are previous question(s) and it/they
431         # haven't been answered correctly
432         if not (self.can_access_question(request.user, question, module_id,
433                                          question_id)):
434             return Response(
435                 {'ans': "Previous question(s) haven't been answered"
436                         + " correctly yet"},
437                 status=status.HTTP_403_FORBIDDEN
438             )
439
440         solved = question.evaluate(request.data["answers"])
441
442         # only saves the points if the question hasn't been answered yet
443         if solved and not question.try_set.filter(
444                 user=request.user, solved=True).exists():
445             request.user.profile.ranking += question.get_points()
446             request.user.profile.save()
447         Try(user=request.user, question=question,
448             answer=str(request.data["answers"]), solved=solved).save()
449         response = {"evaluate": solved}
450         if solved:
451             next_type = ""
452             if not question.is_last_question():
453                 next_type = "question"
454             elif not course_module.is_last_module():
455                 next_type = "module"
```

```
456             elif course.quizquestion_set.exists():
457                 next_type = "quiz"
458             response['next'] = next_type
459             if solved and question.feedback:
460                 # response['custom_feedback'] = question.custom_feedback()
461                 response['feedback'] = question.feedback
462         return Response(response)
463
464
465 class AnswerView(APIView):
466     """
467     Shows all possible answers to a question.
468     :author: Claas Voelcker
469     """
470     authentication_classes = (authentication.TokenAuthentication,)
471     permission_classes = (permissions.IsAuthenticated,)
472
473     def get(self, request, course_id, module_id, question_id, format=None):
474         """
475         Lists the answers for a question
476         """
477         course = Course.objects.get(id=course_id)
478         module = course.module_set.all()[int(module_id)]
479         question = module.question_set.all()[int(question_id)]
480         answers = question.answer_set()
481         data = [serializers.get_answer_serializer(answer) for answer in
482                 answers]
483         return Response(data, status=status.HTTP_200_OK)
484
485     def post(self, request, format=None):
486         """
487         Not implemented
488         :param request:
489         :param format:
490         :return:
491         """
492         return Response({"ans": 'Method not allowed'},
493                         status=status.HTTP_405_METHOD_NOT_ALLOWED)
494
495
496 def calculate_quiz_points(old_percentage, new_percentage, difficulty):
497     """
498     calculates the quiz points from the old existing statistics and the new
499     quiz answers
500     :param old_percentage: percentage of questions already answered correctly
501     :param new_percentage: percentage of questions newly answered correctly
502     :param difficulty: the course difficulty
503     :return: the additional points
504     """
505     multiplier = 2 if difficulty == 2 else 1
506     ranking_threshold = [0.4, 0.7, 0.9]
507     old_extra_points = [x[0] for x in enumerate(ranking_threshold) if
508                         x[1] > old_percentage]
509     new_extra_points = [x[0] for x in enumerate(ranking_threshold) if
510                         x[1] > new_percentage]
511     old_extra_points = 3 if not old_extra_points else old_extra_points[0]
512     new_extra_points = 3 if not new_extra_points else new_extra_points[0]
513     return max(0, 5 * multiplier * (new_extra_points - old_extra_points))
514
515
516 class QuizView(APIView):
517     """
518     Shows the quiz question of the current course in get
519     evaluates this quiz question in post
520     @author Leonhard Wiedmann
521     """
```

```
522     authentication_classes = (authentication.TokenAuthentication,)
523     permission_classes = (permissions.IsAuthenticated,)
524
525     def get(self, request, course_id):
526         """
527         Shows the current quiz question if it exists.
528         When this id does not exist throws error message
529         """
530         course = Course.objects.filter(id=course_id).first()
531
532         # check if user did last question of the last module
533         # if valid the course is completed
534         module = course.module_set.all()[len(course.module_set.all()) - 1]
535         question = module.question_set.all(
536         )[len(module.question_set.all()) - 1]
537         if not Try.objects.filter(question=question, solved=True).exists():
538             return Response({"error": "complete the course first"},
539                             status=status.HTTP_403_FORBIDDEN)
540
541         quiz = course.quizquestion_set.all()
542         if len(quiz) in range(5, 21):
543             quiz = serializers.QuizSerializer(quiz, many=True)
544
545             return Response(quiz.data)
546         return Response({"error": "this quiz is invalid"},
547                         status=status.HTTP_400_BAD_REQUEST)
548
549     def post(self, request, course_id, format=None):
550         """
551         Resolves this quiz question for the current user.
552         """
553         # post does two things (return feedback for questions and whole quiz)
554         # this switch/case differentiates between the two
555         if request.data['type'] == "check_answers":
556             course = Course.objects.get(id=course_id)
557             quiz = course.quizquestion_set.all()
558             all_question_length = len(quiz)
559             if all_question_length <= 0:
560                 return Response({"error": "this quiz does not exist"},
561                                 status=status.HTTP_404_NOT_FOUND)
562             # checks if the submission is wrong (different lengths of the
563             # arrays)
564             if len(quiz) != len(request.data['answers']):
565                 resp = "the quiz has {} question and your evaluation has {}"\
566                     .format(len(quiz), len(request.data['answers']))
567                 return Response({"error": resp, "test": request.data},
568                                 status=status.HTTP_400_BAD_REQUEST)
569             response = []
570             newly_solved = 0
571             old_solved = 0
572             for i, quiz_entry in enumerate(quiz):
573                 answer_solved = request.data['answers'][i]
574                 for answer in request.data['answers']:
575                     if 'id' in answer and quiz_entry.id is answer['id']:
576                         answer.pop('id')
577                         answer_solved = answer
578                         break
579                 solved = quiz_entry.evaluate(answer_solved)
580                 if solved and not quiz_entry.try_set.filter(
581                         user=request.user, solved=True).exists():
582                     newly_solved += 1
583                     request.user.profile.ranking += quiz_entry.get_points()
584                 elif quiz_entry.try_set.filter(user=request.user,
585                                                solved=True).exists():
586                     old_solved += 1
587                 Try(user=request.user, quiz_question=quiz_entry,
```

```
588                     answer=str(request.data), solved=solved).save()
589
590                 response.append({"name": quiz[i].question, "solved": solved})
591
592             old_extra = float(old_solved / all_question_length)
593             new_extra = float(
594                 (newly_solved + old_solved) / all_question_length)
595             request.user.profile.ranking += calculate_quiz_points(
596                 old_extra, new_extra, course.difficulty)
597             request.user.profile.save()
598
599             return Response(response, status=status.HTTP_200_OK)
600         if request.data['type'] == 'get_answers':
601             course = Course.objects.get(id=course_id)
602             quiz_question = course.quizquestion_set.all()[request.data['id']]
603             answers = [answer.id for answer in
604                        quiz_question.quizanswer_set.all() if answer.correct]
605             return Response({'answers': answers}, status.HTTP_200_OK)
606         return Response({'ans': 'Could not process request'},
607                         status.HTTP_400_BAD_REQUEST)
608
609
610 class UserView(APIView):
611     """
612     Shows a user profile
613     @author Claas Voelcker
614     """
615     authentication_classes = (authentication.TokenAuthentication,)
616     permission_classes = (permissions.IsAuthenticated,)
617
618     def get(self, request, user_id=False, format=None):
619         """
620         Shows the profile of any user if the requester is mod,
621         or the profile of the requester
622
623         TODO: If the behaviour that an admin is allowed to receive information
624         about a specific user, will be used again,
625         a custom_permission should be written.
626         """
627         user = request.user
628         if user_id:
629             if user.profile.is_admin():
630                 user = User.objects.filter(id=user_id).first()
631                 if not user:
632                     return Response({'ans': 'User not found'},
633                                     status=status.HTTP_404_NOT_FOUND)
634             else:
635                 raise PermissionDenied(detail=None, code=None)
636
637         user = serializers.UserSerializer(user)
638         return Response(user.data)
639
640     def post(self, request, format=None):
641         """
642         Post is used to update the profile of the requesting user
643         @author Tobias Huber
644         """
645         user = request.user
646         data = request.data
647
648         if 'oldpassword' in data:
649             if not request.user.check_password(request.data['oldpassword']):
650                 return Response({'error': 'given password is incorrect'},
651                                 status=status.HTTP_400_BAD_REQUEST)
652         else:
653             return Response({'error': 'password is required'},
```

```
654                             status=status.HTTP_400_BAD_REQUEST)
655
656         user_serializer = serializers.UserSerializer(user, data=data,
657                                                      partial=True)
658         if user_serializer.is_valid():
659             user_serializer = user_serializer.update(
660                 user,
661                 validated_data=request.data)
662             return Response({'ans': 'Updated user ' + user.username},
663                             status=status.HTTP_200_OK)
664         return Response(user_serializer.errors,
665                         status=status.HTTP_400_BAD_REQUEST)
666
667
668 class UserRegisterView(APIView):
669     """
670     Saves a new user
671     @author Tobias Huber
672     """
673     authentication_classes = []
674     permission_classes = []
675
676     def post(self, request, user_id=False, format=None):
677         """
678         Saves a new user.
679         """
680         if user_id:
681             return Response({'ans': 'Please use the UserView to update data'},
682                             status=status.HTTP_403_FORBIDDEN)
683         user_serializer = serializers.UserSerializer(data=request.data)
684         if user_serializer.is_valid():
685             user_serializer.create(request.data)
686             return Response({'ans': 'Created a new user'},
687                             status=status.HTTP_201_CREATED)
688         return Response(user_serializer.errors,
689                         status=status.HTTP_400_BAD_REQUEST)
690
691
692 class MultiUserView(APIView):
693     """
694     Shows an overview over all users
695     @author Claas Voelcker
696     """
697     authentication_classes = (authentication.TokenAuthentication,)
698     permission_classes = (custom_permissions.IsAdmin,)
699
700     def get(self, request):
701         """
702         Returns all users
703         """
704         users = User.objects.all()
705         data = serializers.UserSerializer(users, many=True).data
706         return Response(data)
707
708     def post(self, request, format=None):
709         """
710         Not implemented
711         """
712         return Response({'ans': 'Method not allowed'},
713                         status=status.HTTP_405_METHOD_NOT_ALLOWED)
714
715
716 class StatisticsView(APIView):
717     """
718     A class displaying statistics information for a given user. It is used to
719     access the try object.
```

```
720     @author: Claas Voelcker
721     """
722     authentication_classes = (authentication.TokenAuthentication,)
723     permission_classes = (permissions.IsAuthenticated,)
724 
725     def get(self, request, user_id=None):
726         """
727         shows the statistics of the given user
728         """
729         user = request.user if not user_id else User.objects.get(id=user_id)
730         tries = Try.objects.filter(user=user)
731         data = serializers.TrySerializer(tries, many=True).data
732         return Response(data)
733 
734     def post(self, request, format=None):
735         """
736         implements filtering logic for the statistics
737         :param request:
738         :param format:
739         :return:
740         """
741         import time
742         import csv
743         data = request.data
744         user = request.user
745 
746         tries = Try.objects.all()
747 
748         groups = user.groups.values_list('name', flat=True)
749 
750         is_mod = 'moderator' in groups or 'admin' in groups
751 
752         # the simplest call is if the user just wants its statistic
753         if 'id' in data and data['id'] == user.id:
754             tries = tries.filter(user=user)
755 
756         # A moderator can get all statistics of his created courses
757         # with 'get_courses' as in put the it will return all courses created
758         # by this user
759         elif is_mod and 'course' in data and 'admin' not in groups:
760             tries = tries.filter(
761                 question__module__course__responsible_mod=user)
762 
763         # admins can get all statistics of all users
764         elif 'admin' in groups:
765             if 'id' in data:
766                 tries.filter(user__id=data['id'])
767         else:
768             return Response({'error': 'invalid userID'},
769                             status=status.HTTP_403_FORBIDDEN)
770 
771         # return all statistics after prefiltering for this course
772         if 'course' in data:
773 
774             tries = tries.filter(question__module__course__id=data['course'])
775 
776             if 'list_questions' in data:
777                 course = Course.objects.filter(id=data['course']).first()
778                 value = []
779                 index = 0
780                 for module in course.module_set.all():
781                     value.append([])
782                     for question in module.question_set.all():
783                         value[index].append({'name': question.title,
784                                              'solved': len(
785                                                  question.try_set.filter(
```

```
                                                    solved=True).all()),
                                            'not solved': len(
                                                question.try_set.filter(
                                                    solved=False).all())})
                    index += 1
                return Response(value)

        # get the statistics for a specific time
        if ('date' in data
                and 'start' in data['date']
                and 'end' in data['date']):
            start = data['date']['start']
            end = data['date']['end']
            tries = tries.filter(
                date__range=[start, end])

        # filter just for solved tries
        if 'solved' in data:
            tries = tries.filter(solved=data['solved'])

        # filter for a specific category
        if 'category' in data:
            tries = tries.filter(
                question__module__course__category__name=data['category'])

        # if this variable is set the view will return a array of dicts which
        # are {name: string, color: string, counter: number}
        if 'categories__with__counter' in data:
            categories = CourseCategory.objects.all()
            value = []
            for cat in categories:
                value.append(
                    {
                        'name': cat.name,
                        'color': cat.color,
                        'counter': len(tries.filter(
                            question__module__course__category=cat))
                    })
            return Response(value)

        serialize_data = None

        # filters the statistics and counts for the 'filter' variable
        if 'filter' in data:
            value = {}
            for trie in tries:
                if not str(getattr(trie, data['filter'])) in value:
                    value[str(getattr(trie, data['filter']))] = 1
                else:
                    value[str(getattr(trie, data['filter']))] += 1
            return Response(value)

        # this part orders the list for the 'order' value in the request
        if 'order' in data:
            tries = tries.order_by(data['order'])

        if 'serialize' in data:
            serialize_data = serializers.TrySerializer(tries, many=True,
                                                       context={
                                                           'serialize': data[
                                                               'serialize']}).data
        else:
            serialize_data = serializers.TrySerializer(tries, many=True).data

        if 'format' in data and data['format'] == 'csv':
            response = HttpResponse(content_type='text/csv')
```

```
            filename = time.strftime('%d/%m/%Y') + '-' + user.username + '.csv'
            content = 'attachment; filename="' + filename
            response['Content-Disposition'] = content
            writer = csv.writer(response)
            writer.writerow(['question', 'user', 'date', 'solved'])
            for row in serialize_data:
                profile = Profile.objects.get(user__username=row['user'])
                profile_hash = profile.get_hash()
                writer.writerow(
                    [row['question'],
                     profile_hash,
                     row['date'],
                     row['solved']])
            return response
        return Response(serialize_data)


class RankingView(APIView):
    """
    A view for the ranking. The get method returns an ordered list of all users
    according to their rank.
    """
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (permissions.IsAuthenticated,)

    def get(self, request, format=None):
        """
        API request for ranking information
        :param request: can be empty
        :param format: request: can be empty
        :return: a json response with ranking information
        """
        profiles = Profile.objects.all().reverse()
        data = serializers.RankingSerializer(profiles).data
        return Response(data)

    def post(self, request, format=None):
        """
        Not implemented
        """
        return Response({'ans': 'Method not allowed'},
                        status=status.HTTP_405_METHOD_NOT_ALLOWED)


class RequestView(APIView):
    """
    The RequestView class is used to submit a request for moderator rights.

    The request can be accessed via "clonecademy/user/request/"
    @author Tobias Huber
    """
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (permissions.IsAuthenticated,)

    def get(self, request, format=None):
        """
        Returns True if request is allowed and False if request isn't allowed
        or the user is already mod.
        """
        allowed = (not request.user.profile.is_mod()
                   and request.user.profile.modrequest_allowed())
        return Response({'allowed': allowed},
                        status=status.HTTP_200_OK)

    def post(self, request, format=None):
        """
```

```
        Handels the moderator rights request. Expects a reason and extracts the
        user from the request header.
        """
        data = request.data
        user = request.user
        profile = user.profile
        if not user.profile.modrequest_allowed():
            return Response(
                {'ans': 'User is mod or has sent too many requests'},
                status=status.HTTP_403_FORBIDDEN)
        # pay attention because there could be localization errors
        profile.last_modrequest = timezone.now()
        profile.save()
        send_mail(
            'Moderator rights requested by {}'.format(user.username),
            'The following user {} requested moderator rights for the'
            'CloneCademy platform. \n'
            'The given reason for this request: \n{}\n '
            'If you want to add this user to the moderator group, access the '
            'profile {} for the confirmation field.\n '
            'Have a nice day,\n your CloneCademy bot'.format(
                user.username, data['reason'],
                user.profile.get_link_to_profile()),
            'bot@clonecademy.de',
            [admin.email for
             admin in Group.objects.get(name='admin').user_set.all()]
        )
        return Response({'Request': 'ok'}, status=status.HTTP_200_OK)


class UserRightsView(APIView):
    """
    Used to promote or demote a given user (by id)

    This View is used to grant or revoke specific rights (user|moderator|admin)
    The POST data must include the following fields
    {"right": "moderator"|"admin",
    "action": "promote"|"demote"}.
    Returns the request.data if validation failed.

    The user_id is to be provided in the url.

    TODO: try the generic.create APIView. Its behaviour isn't really different
    from the current. It just provides additional success-headers in a way
    I do not understand.
    """

    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (custom_permissions.IsAdmin,)

    def post(self, request, user_id, format=None):
        """
        changes the group membership of the user
        """
        data = request.data
        right_choices = ['moderator', 'admin']
        action_choices = ['promote', 'demote']
        errors = {}

        # validation
        if not data['right'] or not data['right'] in right_choices:
            errors['right'] = ('this field is required and must be one of '
                               + 'the following options'
                               + ', '.join(right_choices))
        if not data['action'] or not data['action'] in action_choices:
            errors['action'] = ('this field is required and must be one of '
```

```
                                + 'the following options'
                                + ', '.join(action_choices))
        if not User.objects.filter(id=user_id).exists():
            errors['id'] = 'a user with the id #' + user_id + ' does not exist'
        if errors:
            return Response(errors, status=status.HTTP_400_BAD_REQUEST)

        # actual behaviour
        user = User.objects.get(id=user_id)
        group = Group.objects.get(name=data['right'])
        action = data['action']
        if action == 'promote':
            user.groups.add(group)
        elif action == 'demote':
            user.groups.remove(group)
        return Response(serializers.UserSerializer(user).data)

    def get(self, request, user_id, format=None):
        """
        This API is for debug only.
        It comes in quite handy with the browsable API
        """
        user = User.objects.get(id=user_id)
        return Response({'username': user.username,
                         'is_mod?':
                             user.groups.filter(name='moderator').exists(),
                         'is_admin?':
                             user.groups.filter(name='admin').exists()})


class PwResetView(APIView):
    """
    Resets the password of a user and sends the new one to the email adress
    of the user

    {
        "email": the email of the user
    }
    """

    authentication_classes = ()
    permission_classes = ()

    def post(self, request, format=None):
        """
        Sends a mail to the user containing a new one time password
        :param request:
        :param format:
        :return:
        """
        data = request.data
        if 'email' not in data:
            return Response({'ans': 'you must provide an email'},
                            status=status.HTTP_400_BAD_REQUEST)
        elif not User.objects.filter(email=data['email']).exists():
            return Response({'ans': 'no user with email: ' + data['email']},
                            status=status.HTTP_404_NOT_FOUND)

        # if request data is valid:
        user = User.objects.get(email=data['email'])
        # generate a random password with the rand() implementation of
        # django.utils.crypto
        new_password = get_random_string(length=16)

        send_mail(
            'Password Reset on clonecademy.net',
```

```
1050            ('Hello {},\n \n'
1051             + 'You have requested a new password on clonecademy.net \n'
1052             + 'Your new password is: \n{} \n \n'
1053             + 'Please change it imediately! \n'
1054             + 'Have a nice day,\nyour CloneCademy bot').format(
1055                 user.username, new_password),
1056            'bot@clonecademy.de',
1057            [user.email]
1058        )
1059        user.set_password(new_password)
1060        user.save()
1061        return Response(status=status.HTTP_200_OK)
```