Coverage for **django/learning_base/serializers.py** : 89%

299 statements | 267 run | 32 missing | 0 excluded

```
"""
module containing all serializers
"""

from django.contrib.auth.models import User

from rest_framework import serializers
from rest_framework.exceptions import ParseError

from .info.serializer import InformationYoutubeSerializer, \
    InformationTextSerializer
from .multiple_choice.serializer import \
    MultipleChoiceQuestionSerializer
from .models import Question, CourseCategory, Module, Course, QuizQuestion, \
    QuizAnswer, LearningGroup, Try, Profile


def get_answer_serializer(obj):
    """
    dispatch function that chooses the correct serializer
    :param obj: the answers object to be serialiaized
    :return: a json serialization
    """
    serializer = obj.get_serializer()
    return serializer(obj).data


class QuestionSerializer(serializers.ModelSerializer):
    """
    The serializer responsible for the Question object
    :author: Claas Voelcker
    """

    class Meta:
        """
        Meta information (which fields are serialized for the representation)
        """
        model = Question
        fields = ('title', 'text', 'feedback',)

    def to_representation(self, obj):
        """
        Appends additional information to the model.
        :param: obj: The object that should be serialized (Question)
        :return: value: a valid json object containing all required fields
        """
        course_module = obj.module
        value = super(QuestionSerializer, self).to_representation(obj)
        value['type'] = obj.__class__.__name__
        user = self.context['request'].user

        # calculate the current progress of the user in a array of arrays
        # the outer array is the module and the inner
        # is the title of the quesiton
        # e.g [['question 1', 'question, 2'], ['quesiton 3']]
        value['progress'] = []
        answered_question_before = True
        for course_module in obj.module.course.module_set.all():
            module_set = []
```

```
60              for question in course_module.question_set.all():
61                  if answered_question_before and question.try_set.filter(
62                          solved=True, user=user).exists():
63                      module_set.append({'solved': True, 'title': question.title})
64
65                  else:
66                      answered_question_before = False
67                      module_set.append({'solved': False, 'title': question.title})
68              value['progress'].append(module_set)
69
70          value['last_question'] = obj.is_last_question()
71          value['last_module'] = course_module.is_last_module()
72          value['learning_text'] = course_module.learning_text
73          serializer = obj.get_serializer()
74          value['question_body'] = serializer(obj).data
75
76          value['solved'] = obj.try_set.filter(solved=True, user=user).exists()
77
78          return value
79
80      def create(self, validated_data):
81          """
82          Serializer that governs the dispatch to specific class serializers
83          :param validated_data: the data to be serialized
84          :return: serialized representation
85          """
86          question_type = validated_data.pop('type')
87          if question_type == 'multiple_choice':
88              MultipleChoiceQuestionSerializer().create(validated_data)
89          elif question_type == 'info_text':
90              InformationTextSerializer().create(validated_data)
91          elif question_type == 'info_text_youtube':
92              InformationYoutubeSerializer().create(validated_data)
93          else:
94              raise ParseError(
95                  detail='{} is not a valid question type'.format(
96                      question_type))
97
98
99  class QuestionEditSerializer(serializers.ModelSerializer):
100     """
101     The serializer to get all information to edit a question
102     :author: Leonhard Wiedmann
103     """
104     class Meta:
105         model = Question
106         fields = ('title', 'text', 'id', 'feedback')
107
108     def to_representation(self, obj):
109         """
110         to json representation serialization
111         :param obj: the object to be serialized
112         :return: a serialized representation
113         """
114         value = super(QuestionEditSerializer, self).to_representation(obj)
115         value['type'] = obj.__class__.__name__
116         serializer = obj.get_edit_serializer()
117         value['question_body'] = serializer(obj).data
118         return value
119
120
121 class CourseCategorySerializer(serializers.ModelSerializer):
122     """
123     A serializer for course categories
124     :author: Claas Voelcker
```

```
125         """
126         class Meta:
127             model = CourseCategory
128             fields = ('name', 'color', 'id',)
129 
130         color = serializers.RegexField(r'^#[a-fA-F0-9]{6}', max_length=7,
131                                        min_length=7, allow_blank=False)
132 
133 
134 class ModuleEditSerializer(serializers.ModelSerializer):
135     """
136     A serializer to get module editing data
137     :auhor: Leonhard Wiedmann
138     """
139     class Meta:
140         model = Module
141         fields = ('name', 'id', 'learning_text', 'order')
142 
143     def to_representation(self, obj):
144         """
145         responsible for returning a valid json response
146         :param obj: the object to be serialized
147         :return: a valid json representation
148         """
149         value = super(ModuleEditSerializer, self).to_representation(obj)
150 
151         questions = obj.question_set.all()
152         value['questions'] = QuestionEditSerializer(questions, many=True).data
153         return value
154 
155 
156 class ModuleSerializer(serializers.ModelSerializer):
157     """
158     The serializer for modules
159     :author: Leonhard Wiedmann
160     """
161     class Meta:
162         model = Module
163         fields = ('name', 'learning_text', 'id')
164 
165     def to_representation(self, obj):
166         """
167         This function makes the serialization and is needed to correctly
168         display nested objects.
169         :param obj: a
170         :return:
171         """
172 
173         value = super(ModuleSerializer, self).to_representation(obj)
174 
175         questions = Question.objects.filter(module=obj)
176         questions = QuestionSerializer(
177             questions, many=True, read_only=True, context=self.context).data
178 
179         value['questions'] = questions
180         return value
181 
182     def create(self, validated_data):
183         """
184         This method is used to save modules and their respective questions
185         """
186         questions = validated_data.pop('questions')
187 
188         if questions is None or len(questions) is 0:
189             raise ParseError(detail='empty module is not allowed', code=None)
```

```
190
191         module = Module(**validated_data)
192         module.course = validated_data['course']
193         module.save()
194
195         question_id = []
196         # create a array with the ids for all questions of this module
197         for quest in questions:
198             if 'id' in quest:
199                 question_id.append(quest['id'])
200
201         # check if this is a edit or creation of a new module
202         if question_id:
203             # get all questions for the current module
204             query_questions = Question.objects.filter(
205                 module_id=validated_data['id'])
206             # iterate over the questions and if the questions does not exist in
207             # the edited module it will be removed
208             for question in query_questions:
209                 if question.id not in question_id:
210                     question.delete()
211
212         for question in questions:
213             question['module'] = module
214             question_serializer = QuestionSerializer(data=question)
215             if not question_serializer.is_valid():
216                 raise ParseError(
217                     detail='Error in question serialization', code=None)
218             else:
219                 question_serializer.create(question)
220
221
222 class CourseSerializer(serializers.ModelSerializer):
223     """
224     A serializer to view courses
225     """
226     category = serializers.StringRelatedField()
227     is_visible = serializers.BooleanField(required=False,
228                                           default=False)
229
230     class Meta:
231         model = Course
232         fields = ('name', 'difficulty', 'id', 'language', 'category',
233                   'is_visible', 'description')
234
235     def to_representation(self, obj):
236         """
237         This function serializes the courses.
238         :param obj: the object to be serialized
239         :return: a json serialization
240         """
241
242         value = super(CourseSerializer, self).to_representation(obj)
243
244         all_modules = obj.module_set.all()
245         modules = ModuleSerializer(all_modules, many=True, read_only=True,
246                                    context=self.context).data
247
248         value['modules'] = modules
249
250         num_questions = 0
251         num_answered = 0
252         count_question = 0
253         count_module = 0
254         for module in modules:
```

```
            count_module += 1
            for question in module['questions']:
                count_question += 1
                if question['solved']:
                    num_answered += 1
                else:
                    value['next_question'] = count_question
                    value['current_module'] = count_module
                num_questions += 1
        value['num_answered'] = num_answered
        value['num_questions'] = num_questions
        value['responsible_mod'] = obj.responsible_mod.id
        return value

    def create(self, validated_data):
        """
        This method is used to save courses together with all modules and
        questions.
        :param validated_data: valid data for the Course object
        """
        modules = validated_data.pop('modules')
        quiz_data = []
        if 'quiz' in validated_data:
            quiz_data = validated_data.pop('quiz')

        # check if course is empty and raise error if so
        if not modules:
            raise ParseError(detail='Course needs to have at least one module',
                             code=None)
        category = validated_data.pop('category')
        category = CourseCategory.objects.get(name=category)
        validated_data['category'] = category
        course = Course(**validated_data)
        course.save()

        # add quiz to a course
        if quiz_data and len(quiz_data) >= 5:
            try:
                if 5 <= len(quiz_data) <= 20:
                    quiz_id = [q['id'] for q in quiz_data if 'id' in q.keys()]
                    for quiz in course.quizquestion_set.all():
                        if quiz.id not in quiz_id:
                            quiz.delete()
                    for quiz in quiz_data:
                        quiz_serializer = QuizSerializer(data=quiz)
                        if not quiz_serializer.is_valid():
                            raise ParseError(
                                detail=str(quiz_serializer.errors),
                                code=None)
                        else:
                            quiz['course'] = course
                            quiz_serializer.create(quiz)
            except ParseError as error:
                if 'id' not in validated_data:
                    course.delete()
                raise ParseError(detail=error.detail, code=None)
        else:
            for quiz in course.quizquestion_set.all():
                quiz.delete()

        # create a array with the ids for all module ids of this course
        module_id = []
        for course_module in modules:
            if 'id' in course_module:
                module_id.append(course_module['id'])
```

```
320
321         # check if this is a edit or creation of a new course
322         if module_id:
323             # get all modules for the current course
324             module_query = Module.objects.filter(
325                 course_id=validated_data['id'])
326             # iterate over the modules and if the modules does not exist in the
327             # edited course it will be removed
328             for course_module in module_query:
329                 if course_module.id not in module_id:
330                     course_module.delete()
331
332         try:
333             if len(modules) <= 0:
334                 raise ParseError(detail='no Empty course allowed', code=None)
335             for module in modules:
336                 module_serializer = ModuleSerializer(data=module)
337                 if not module_serializer.is_valid():
338                     raise ParseError(
339                         detail='Error in module serialization', code=None)
340                 else:
341                     module['course'] = course
342                     module_serializer.create(module)
343             return True
344         except ParseError as error:
345             if 'id' not in validated_data:
346                 course.delete()
347             raise ParseError(detail=error.detail, code=None)
348
349
350 class CourseEditSerializer(serializers.ModelSerializer):
351     """
352     A serializer that returns all data needed to edit the course
353     :author: Leonhard Wiedmann
354     """
355     category = serializers.StringRelatedField()
356
357     class Meta:
358         model = Course
359         fields = (
360             'name', 'id', 'category', 'difficulty', 'language',
361             'responsible_mod',
362             'is_visible', 'description')
363
364     def to_representation(self, obj):
365         """
366         method responsible for serializing a Curse object for editing purposes
367         :param obj: the object to be serialized
368         :return: a json representation of the object
369         """
370         value = super(CourseEditSerializer, self).to_representation(obj)
371         all_modules = obj.module_set.all()
372         modules = ModuleEditSerializer(all_modules, many=True).data
373         value['modules'] = modules
374
375         all_quiz = obj.quizquestion_set.all()
376         quiz = QuizSerializer(all_quiz, many=True, context={'edit': True}).data
377         value['quiz'] = quiz
378
379         return value
380
381
382 class QuizSerializer(serializers.ModelSerializer):
383     """
384     Quiz Serializer for a single quiz question
```

```
385     :author: Leonhard Wiedmann
386     """
387
388     class Meta:
389         model = QuizQuestion
390         fields = ('question', 'image', 'id',)
391
392     def create(self, validated_data):
393         """
394         method creating a Quiz object form a json input
395         :param validated_data: validated json data for the object
396         """
397         if 'answers' not in validated_data:
398             raise ParseError(detail='The quiz has no answers',
399                              code=None)
400         answers = validated_data.pop('answers')
401         quiz = QuizQuestion(**validated_data)
402         quiz.save()
403         try:
404             if len(answers) != 4:
405                 raise ParseError(detail='Quiz must have 4 answers', code=None)
406             for ans in answers:
407                 quiz_answer_serializer = QuizAnswerSerializer(data=ans)
408                 if not quiz_answer_serializer.is_valid():
409                     raise ParseError(detail=str(quiz_answer_serializer.errors),
410                                      code=None)
411                 else:
412                     ans['quiz'] = quiz
413                     quiz_answer_serializer.create(ans)
414             if not quiz.is_solvable():
415                 raise ParseError(detail='This quiz is not solvable', code=None)
416         except ParseError as error:
417             quiz.delete()
418             raise ParseError(detail=error.detail, code=None)
419
420     def to_representation(self, obj):
421         """
422         representation of a Quiz object
423         :author: Claas Voelcker
424         :param obj: the user object to be serialized
425         :return: a json representation of the object
426         """
427         value = super(QuizSerializer, self).to_representation(obj)
428         value['answers'] = QuizAnswerSerializer(obj.answer_set(), many=True,
429                                                 context=self.context).data
430         return value
431
432
433 class QuizAnswerSerializer(serializers.ModelSerializer):
434     """
435     Quiz Answer Serializer
436     :author: Leonhard Wiedmann
437     """
438
439     class Meta:
440         model = QuizAnswer
441         fields = ('text', 'img', 'id')
442
443     def create(self, validated_data):
444         """
445         method creating a QuizAnswer object form a json input
446         :param validated_data: validated json data for the object
447         """
448         quiz_answer = QuizAnswer(**validated_data)
449         quiz_answer.save()
```

```
450
451     def to_representation(self, obj):
452         """
453         representation of a Quiz object
454         :author: Claas Voelcker
455         :param obj: the user object to be serialized
456         :return: a json representation of the object
457         """
458         value = super(QuizAnswerSerializer, self).to_representation(obj)
459         if 'edit' in self.context and self.context['edit']:
460             value['correct'] = obj.correct
461         return value
462
463
464 class GroupSerializer(serializers.ModelSerializer):
465     """
466     Model serializer for the Group model
467     :author: Claas Voelcker
468     """
469
470     class Meta:
471         model = LearningGroup
472         fields = ('name', 'id')
473
474
475 class UserSerializer(serializers.ModelSerializer):
476     """
477     Model serializer for the User model
478     :author: Claas Voelcker
479     """
480
481     groups = serializers.StringRelatedField(many=True)
482
483     class Meta:
484         model = User
485         fields = (
486             'username', 'email', 'id', 'date_joined', 'groups', 'first_name',
487             'last_name')
488
489     def to_representation(self, obj):
490         """
491         representation of a user object
492         :author: Leonhard Wiedmann
493         :param obj: the user object to be serialized
494         :return: a json representation of the object
495         """
496         value = super(UserSerializer, self).to_representation(obj)
497
498         if 'language' not in value:
499             value['language'] = 'en'
500         profile = Profile.objects.filter(user=obj).first()
501         value['language'] = profile.language
502
503         value['avatar'] = profile.avatar
504         value['ranking'] = profile.ranking
505         return value
506
507     def create(self, validated_data):
508         """
509         method creating a User object form a json input
510         :author: Leonhard Wiedmann
511         :param validated_data: validated json data for the object
512         """
513         profile_data = validated_data.pop('profile')
514         validated_data.pop('groups')
```

```
515         if 'language' not in validated_data:
516             validated_data['language'] = 'en'
517         profile_data['language'] = validated_data.pop('language')
518         if 'avatar' in validated_data:
519             profile_data['avatar'] = validated_data.pop('avatar')
520         # if 'language' in profile_data:
521         #    profile_data['language'] = validated_data.pop('language')
522         user = User.objects.create_user(**validated_data)
523         profile = Profile(user=user, **profile_data)
524         profile.save()
525 
526     def update(self, instance, validated_data):
527         """
528         Updates a given user instance
529         Note: Only updates fields changeable by user
530         :author: Tobias Huber
531         :param validated_data: validated data for the input
532         """
533         instance.email = validated_data['email']
534         instance.first_name = validated_data['first_name']
535         instance.last_name = validated_data['last_name']
536         if 'password' in validated_data:
537             instance.set_password(validated_data['password'])
538         profile = instance.profile
539         # profile.language = validated_data['language']
540         profile.avatar = validated_data['avatar']
541         profile.save()
542         instance.save()
543 
544 
545 class TrySerializer(serializers.ModelSerializer):
546     """
547     Model serializer for the Try model
548     :author: Claas Voelcker
549     """
550     user = serializers.StringRelatedField()
551     question = serializers.StringRelatedField()
552     date = serializers.DateTimeField(format='%d/%m/%Y')
553 
554     class Meta:
555         model = Try
556         fields = ('user', 'question', 'date', 'solved')
557 
558     def to_representation(self, obj):
559         """
560         converts try objects to a serialized version
561         :param obj: the object to be serialized
562         :return: a json serialization
563         """
564         data = super(TrySerializer, self).to_representation(obj)
565 
566         if 'serialize' in self.context:
567             for serial in self.context['serialize']:
568                 value = obj
569                 for child in serial.split('__'):
570                     if value is None:
571                         break
572                     value = getattr(value, child)
573                 data[serial] = value
574         return data
575 
576 
577 class RankingSerializer(serializers.BaseSerializer):
578     """
579     A serializer for all rankings
```

```
580         :author: Claas Voelcker
581         """
582
583         def to_representation(self, instance):
584             """
585             a serialization of profile rankings
586             :param instance: an ordered profile list
587             :return: a dictionary with ranking information
588             """
589             value = []
590             for profile in instance:
591                 value.append({
592                     'name': profile.user.username,
593                     'id': profile.id,
594                     'ranking': profile.ranking
595                 })
596             return value
```