About Django's model- and form validations

Submitted by schneck on Fr, 07/01/2011 - 11:38

I’ve had some trouble with validation in django, primarily because you can either perform model- or form validation, which seem to interfere in some way. Validation is often a problem for django-newbies (and experienced developers as well), but first of all, the reason for this is not because it does not work properly, but because it’s explained – and therefore used – in a confusing way. In this blog post I’ll try to put together validation aspects and explain it in an understandable fashion.

Here is our test model which has only one field. This field is intended to be mandatory, and the model should never be saved with having it empty:

```
1. class TestModel(models.Model):
2.     foo = models.CharField(max_length=20, blank=False,
   null=False)
```

Additionally, I built a simple model form for it:

```
1. class TestForm(admin.ModelForm):
2.   class Meta:
3.       model = TestModel
```

The first surprise is, that Django just saves an instance with empty foo-field:

```
1. In [8]: obj = TestModel()
2. In [9]: obj.save()
3. In [10]: TestModel.objects.all()
4. Out[10]: [<TestModel: TestModel object>]
```

You find the reason explained in the documentation: Do not use “null” in string fields.

> *Quote: “Note that empty string values will always get stored as empty strings, not as NULL.”.*

Okay then, this means that it’s impossible to avoid leaving instances with invalid data in the database, if you don’t validate manually. The reason for this is that the “blank”-attribute, which is set in a field declaration, is totally ignored when saving a model if you don’t save it using a form (while it’s validated if you call full_clean(), what we see later). Therefore, the only way around is, to use a form:

```
1. In [9]: from testapp.forms import TestForm
2. In [10]: f = TestForm(instance=obj)
3. In [11]: f.is_valid()
4. Out[11]: False
```

Well, so far it could appear somewhat inconsistent for someone not knowing Django, but after reading the documentation paragraph about storing empty strings, you can work with it anyhow. Now we change our Test-Model and add an foreign field pointing to a new model, which should never be null in the database. The model-file looks this now:

```
1. class RelatedModel(models.Model):
2.     foo = models.CharField(max_length=20, blank=True, null=True,
   default='value')
3.
4. class TestModel(models.Model):
5.     foo = models.CharField(max_length=20, blank=False,
   null=False)
6.     related_model = models.ForeignKey(RelatedModel,
   null=False,blank=False)
```

Now, we try to create a instance of testmodel leaving both fields empty:

```
1. In [1]: obj = TestModel()
2. In [2]: obj.save()
3. …
4. IntegrityError: (1048, "Column 'related_model_id' cannot be
   null")
```

We try to make it through a form:

```
1. In [5]: f = TestForm(instance=obj)
2. In [6]: f.is_valid()
3. Out[6]: False
```

Alright, everything as expected. So there seems only the be a (known and intended – read the paragraph about null-fields again) issue in validating the CharField, which must not be empty.
We implement a custom model-validation now, because we're planning not to use only the contrib-admin, but also to offer an API and a frontend. At least in the API, we do not use forms, but plain object instances, and validation must work in all use cases keeping DRY in mind. This is our test model now:

```
1. class TestModel(models.Model):
2.     foo = models.CharField(max_length=20, blank=False,
   null=False)
3.     related_model = models.ForeignKey(RelatedModel,
   null=False,blank=False)
4.
5.     def clean(self):
6.         if not self.foo:
7.             raise ValidationError('foo must not be empty')
```

What you may expect now is, that when you try to save the instance, it throws an exception because we **did** implement a validation method.

```
1. In [3]: r = RelatedModel()
2. In [4]: r.save()
3. In [5]: obj = TestModel(related_model=r)
4. In [6]: obj.save()
```

It saves. Confusing for a new Django developer. The answer for this can be found – of course – in the Django documentation : clean() is called by full_clean(), but full_clean() is **neved** called

automatically.

> *Quote: "Note that full_clean() will not be called automatically when you call your model's save() method, nor as a result of ModelForm validation. You'll need to call it manually when you want to run model validation outside of a ModelForm."*

Unfortunetaly, this paragraph is a bit confusing. Indeed, Form.is_valid() does not call full_clean(), but it calls clean().

Well, this means that every time we save a model instance outside a form (e.g. in an API handler), we have to call full_clean() first. Okay, lavish, but this is the way it works:

```
1. In [7]: obj.full_clean()
2. …
3. ValidationError: {'foo': [u'This field cannot be blank.'],
   '__all__': [u'foo must not be empty']}
```

This is something we can work with. So far, we implemented a model-based validation for a string field which Django does not validate itself (of course, by adjusting max_length on the foo field, django would validate it. This is just an example). Being happy, we put together a view, which renders a form and saves it after submitting. We test it with empty values:

```
1. In [9]: f = TestForm()
2. In [10]: f.is_valid()
3. Out[10]: False
```

Ok. Now with a valid related field-value:

```
1. In [25]: r = RelatedModel.objects.create()
2. In [26]: f = TestForm({'related_model': r.id})
3. In [27]: f.is_valid()
4. Out[27]: False
5. In [28]: f.errors
6. Out[28]: {'__all__': [u'foo must not be empty'], 'foo': [u'This
   field is required.']}
```

Well, everything's fine: form.is_valid() checks the model's clean()-method. To be really sure this is all fine, we integrate another custom validation and apply all tests again. Here is our new model:

```
1. class TestModel(models.Model):
2.     foo = models.CharField(max_length=20, blank=False,
   null=False)
3.     related_model = models.ForeignKey(RelatedModel,
   null=False,blank=False)
4.
5.     def clean(self):
6.         if not self.foo:
7.             raise ValidationError('foo must not be empty')
8.
9.         if len(self.foo) != 1:
10.            raise ValidationError('foo must be exactly one
   char')
```

And here the tests again:

```
1. In [1]: r = RelatedModel.objects.create()
2. In [2]: obj = TestModel.objects.create(related_model=r,
   foo='bar')
3. In [3]: obj.save()
4. In [4]: # object has been saved, because we did not request a
   validation
5. In [5]: obj.full_clean()
6. …
7. ValidationError: {'all': [u'foo must be exactly one char']}
```

As expected, Django only runs clean() if we request it explicitly. Now, what's about forms?

```
1. In [15]: f = TestForm({'related_model': r.id, 'foo':'bar'})
2. In [16]: f.is_valid()
3. Out[16]: False
4. In [17]: f.errors
5. Out[17]: {'all': [u'foo must be exactly one char']}
6. In [18]: f = TestForm({'related_model': r.id, 'foo':'1'})
7. In [20]: f.is_valid()
8. Out[20]: True
9. In [21]: newobj = f.save()
10. In [22]: newobj
11. Out[22]: <TestModel: TestModel object>
```

Summary

There are a few summary topics that you should keep in mind:

- Form's is_valid() performs model validation automatically.
- Put model validation only in clean() to avoid confusion. Never overwrite full_clean().
- Before you save an object, apply full_clean() on it.
- Do not mix model- and form validation if possible. It just leads to confusion.
- **Do not use form validation if not necessary. Use model validation whenever possible.**

Tags: django, validation