original answers. – HelenM Dec 21, 2017 at 19:50

You can find good answer in the below link. medium.com/@inem.patrick/... – Bijan Mohammadpoor Jan 6, 2021 at 17:30

What is `on_delete=models.DELETE` do? – AnonymousUser Jul 30, 2021 at 6:01

Please edit the question title, on_delete use in OneToOneField and ForeignKey – kamyarmg Mar 2 at 11:12

11 Answers

Sorted by: Highest score (default)

1537

This is the behaviour to adopt when the *referenced* object is deleted. It is not specific to Django; this is an SQL standard. Although Django has its own implementation on top of SQL. **(1)**

There are seven possible actions to take when such event occurs:

- `CASCADE`: When the referenced object is deleted, also delete the objects that have references to it (when you remove a blog post for instance, you might want to delete comments as well). SQL equivalent: `CASCADE`.
- `PROTECT`: Forbid the deletion of the referenced object. To delete it you will have to delete all objects that reference it manually. SQL equivalent: `RESTRICT`.
- `RESTRICT`: *(introduced in Django 3.1)* Similar behavior as `PROTECT` that matches SQL's `RESTRICT` more accurately. (See django documentation example)
- `SET_NULL`: Set the reference to NULL (requires the field to be nullable). For instance, when you delete a User, you might want to keep the comments he posted on blog posts, but say it was posted by an anonymous (or deleted) user. SQL equivalent: `SET NULL`.
- `SET_DEFAULT`: Set the default value. SQL equivalent: `SET DEFAULT`.
- `SET(...)`: Set a given value. This one is not part of the SQL standard and is entirely handled by Django.
- `DO_NOTHING`: Probably a very bad idea since this would create integrity issues in your database (referencing an object that actually doesn't exist). SQL equivalent: `NO ACTION`. **(2)**

Source: Django documentation

See also the documentation of PostgreSQL for instance.

In most cases, `CASCADE` is the expected behaviour, but for every ForeignKey, you should always ask yourself what is the expected behaviour in this situation. `PROTECT` and `SET_NULL` are often useful. Setting `CASCADE` where it should not, can potentially delete all of your database in cascade, by simply deleting a single user.

Additional note to clarify cascade direction

It's funny to notice that the direction of the `CASCADE` action is not clear to many people. Actually, it's funny to notice that **only** the `CASCADE` action is not clear. I understand the cascade behavior might be confusing, however you must think that **it is the same direction as any other action**. Thus, if you feel that `CASCADE` direction is not clear to you, it actually means that `on_delete` behavior is not clear to you.

In your database, a foreign key is basically represented by an integer field which value is the primary key of the foreign object. Let's say you have an entry **comment_A**, which has a foreign key to an entry **article_B**. If you delete the entry **comment_A**, everything is fine. **article_B** used to live without **comment_A** and don't bother if it's deleted. However, if you delete **article_B**, then **comment_A** panics! It never lived without **article_B** and needs it, it's part of its attributes (`article=article_B`, but what is **article_B**???). This is where `on_delete` steps in, to determine how to resolve this *integrity error*, either by saying:

- *"No! Please! Don't! I can't live without you!"* (which is said `PROTECT` or `RESTRICT` in Django/SQL)
- *"All right, if I'm not yours, then I'm nobody's"* (which is said `SET_NULL`)
- *"Good bye world, I can't live without article_B"* and commit suicide (this is the `CASCADE` behavior).
- *"It's OK, I've got spare lover, I'll reference article_C from now"* (`SET_DEFAULT`, or even `SET(...)`).
- *"I can't face reality, I'll keep calling your name even if that's the only thing left to me!"* (`DO_NOTHING`)

I hope it makes cascade direction clearer. :)

Footnotes

> **(1)** Django has its own implementation on top of SQL. And, as mentioned by @JoeMjr2 in the comments below, Django will not create the SQL constraints. If you want the constraints to be ensured by your database (for instance, if your database is used by another application, or if you hang in the database console from time to time), you might want to set the related constraints manually yourself. There is an open ticket to add support for database-level on delete constraints in Django.

> **(2)** Actually, there is one case where `DO_NOTHING` can be useful: If you want to skip Django's implementation and implement the constraint yourself at the database-level.

Share Edit Follow edited Nov 23 at 18:43 answered Jul 15, 2016 at 6:44