django-gdpr-assist Documentation

Release 1.4.2

Wildfish

Jul 12, 2022

Contents:

Tools to help manage user data in the age of GDPR:

- Find, export and anonymise personal data to comply with GDPR requests
- Track anonymisation and deletion of personal data to replay after restoring backups
- Anonymise all models to sanitise working copies of a production database

CHAPTER 1

Installation

Install with:

```
pip install django-gdpr-assist
```

Add to your project's `settings.py`:

```
# Add the app
INSTALLED_APPS = (
    ...
    'gdpr_assist',
    ...
)

# Add a new database to log GDPR actions
DATABASES = {
    ...
    'gdpr_log': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'gdpr-log.sqlite3'),
    },
}
DATABASE_ROUTERS = ['gdpr_assist.routers.EventLogRouter']
```

You'll then need to migrate the new database:

```
./manage.py migrate --database=gdpr_log
```

1.1 Django settings

In addition to the required changes to your settings listed above, there are additional optional settings which you can define to override default behaviour:

1.1.1 `GDPR_PRIVACY_CLASS_NAME = 'PrivacyMeta'`

This allows you to override the default name of the privacy meta class on models.

1.1.2 `GDPR_PRIVACY_INSTANCE_NAME = '_privacy_meta'`

This allows you to override the default name of the instantiated privacy meta class on models.

1.1.3 `GDPR_LOG_DATABASE_NAME = 'gdpr_log'`

The internal name of the log database. You'll need to use this in the `DATABASES` settings, and when migrating.

1.1.4 `GDPR_CAN_ANONYMISE_DATABASE = False`

Set this to `True` to enable the `anonymise_db` management command. You will want this to be `False` on your production deployment.

1.1.5 `GDPR_LOG_ON_ANONYMISE = True`

Set this to `False` to disable entries being created on the fly in the logging database (see `GDPR_LOG_DATABASE_NAME`) during anonymisation, this may be useful for large initial anonyimisation tasks.

By default log entries are created when a instance is anonymised and in bulk when calling the `anonymise_db` command.

If you set this to `False` you can manually create logging for any instance you have anonymised later via `instance._log_gdpr_anonymise()`, handling `post_anonymise` signal or processing over `PrivacyAnonymised` as required i.e a celery queue or cronjob.

1.1.6 `SILENCED_SYSTEM_CHECKS`

By default, gdpr-assist performs migration checks to ensure that you've followed the upgrade instructions correctly to avoid accidental data loss.

See *Upgrading* for more details of the specific checks.

They may cause a slight performance hit to management command which run checks, so while we recommend you leave them on while upgrading, once the upgrade has been completed and succesfully deployed the checks can safely be disabled afterwards by adding them to Django's SILENCED_SYSTEMS_CHECKS setting:

```
SILENCED_SYSTEM_CHECKS = [
    "gdpr_assist.E001",
]
```

CHAPTER 2

Usage

2.1 Configure your models

Define privacy settings in a `PrivacyMeta` class on your model:

```
class MyModel(models.Model):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        blank=True,
        null=True,
        on_delete=gdpr_assist.ANONYMISE(models.SET_NULL),
    )
    display_name = models.CharField(max_length=255)
    private_data = models.IntegerField()
    public_data = models.TextField()

    class PrivacyMeta:
        fields = ['display_name', 'private_data']

        def anonymise_private_data(self, instance):
            return 0

        def search(self, value):
            return self.model.objects.filter(display_name__icontains=value)
```

Next:

- See *Privacy Meta* for the full set of `PrivacyMeta` options, and for how to register a third-party model.
- See *Anonymising* for how anonymisation works.
- See *Admin* to register your model in the admin site, and how to use the admin personal data tool to search and export data.

CHAPTER 3

The PrivacyMeta object

A model needs to be registered with gdpr-assist in order to use any of the anonymisation or export functionality.

The `PrivacyMeta` object tells gdpr-assist which fields are private, and what to do with them.

3.1 Registering automatically

If you define a class called `PrivacyMeta` within your model, gdpr-assist will automatically detect and register your model.

An instance of your `PrivacyMeta` class will then be available on the attribute `_privacy_meta`, the same way a standard `Meta` class works.

For example:

```
class Comment(models.Model):
    name = models.CharField(max_length=255, blank=True)
    age = models.IntegerField(null=True, blank=True)
    message = models.TextField()

    class PrivacyMeta:
        fields = ['name', 'age']

# The following statements are true:
assert(not hasattr(Comment, 'PrivacyMeta'))
assert(hasattr(Comment, '_privacy_meta'))
assert(Comment._privacy_meta.fields == ['name', 'age'])
```

3.2 Registering manually

Sometimes you will want to define your `PrivacyMeta` class somewhere other than on the model - for example when you want to be able to export or anonymise a third-party object, or if you have a particularly complex privacy meta definition and want to store it in a separate file for clarity.

The `gdpr_assist.register(<ModelClass>, [<PrivacyMetaClass>])` function will let you manually register the model with an optional `PrivacyMeta` class.

For example:

```
from django.contrib.auth.models import User

class UserPrivacyMeta:
    fields = ['first_name', 'last_name', 'email']

gdpr_assist.register(User, UserPrivacyMeta, gdpr_default_manager_name="objects_
↪anonymised")
```

If you omit the privacy meta class, one will be generated for you with the default attributes.

Note that `gdpr_default_manager_name` is optional and by default `objects` will be cast to a PrivacyManager, except in the case of Models in which their manager users use_in_migrations, as the User example above does. In these cases a alternate name must be provided and user for queryset anonymisation in order not to create migrations for third parties. `Model.anonymisable_manager()` can also be used to access the PrivacyManager regardless of `gdpr_default_manager_name`.

3.3 Attributes

The `PrivacyMeta` object can have the following attributes:

3.3.1 `can_anonymise = Boolean`

default: True

Whether or not gdpr-assist is used to anonymise the data for this model, if `False`, you can still search and export using gdpr-assist.

3.3.2 `fields = [...]`

List of the names of fields which contain personal information.

These will be the ones which are anonymised; other fields will be unmodified.

Example:

```
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        fields = ['name']
```

3.3.3 `anonymise_<field_name>(self, instance)`

Custom function to anonymise the named field, for when the standard anonymisers won't produce the desired result. This should also be used for custom field types.

Field name must appear in the `fields` list.

It should not return a value; instead it should operate directly on the instance.

Example:

```python
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        def anonymise_name(self, instance):
            instance.name = 'Anon'
```

3.3.4 search_fields = [...]

List of fields to examine when searching for a value in the personal data tool in the admin site.

These field names will be used to build case-insensitive exact matches unless the field name contains a double underscore, __. For example:

- `name` will create a filter of `name__iexact=term`
- `name__icontains` will create a filter of `name__icontains=term`
- `person__name` will create a filter of `person__name=term`

Example:

```python
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        search_fields = ['name__icontains']
```

3.3.5 qs = search(self, value)

Function called by the personal data tool in the admin site, to search the model for the value.

The argument `self` will be a reference to the `PrivacyMeta` instance.

The default function will use `search_fields`, but this can be overridden to perform a custom search.

Should return a queryset (or iterable of objects).

Example:

```python
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        def search(self, value):
            return self.model.objects.filter(name=value.lower())
```

3.3.6 export_fields = [...]

List of fields to export. By default will export all fields.

Example:

```
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        export_fields = ['name']
```

3.3.7 `export_exclude = [...]`

List of fields to not export. By default will exclude foreign keys and many to many fields.

If a field is specified in both `export_fields` and `export_exclude`, it will be excluded.

Example:

```
class MyModel(models.Model):
    name = models.CharField(max_length=255)
    post_count = models.IntegerField(default=0)

    class PrivacyMeta:
        export_exclude = ['post_count']
```

3.3.8 `export_filename = None`

The filename to use for this model when exporting records from it. This should include the `.csv` extension, eg `export_filename = 'user_records.csv'`

If not set, it will default to `<app_name>.<object_name>.csv`, eg `my_app.MyModel.csv`

Example:

```
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        export_filename = 'my_model.csv'
```

3.3.9 `data = export(self, instance)`

Function called by the personal data tool in the admin site, to export a model instance.

By default will export all fields specified in `export_fields` and not excluded by `export_exclude`. They will all be cast to a string.

The default exporter cannot export foreign keys or many to many fields.

Should return a dict.

Example:

```
class MyModel(models.Model):
    name = models.CharField(max_length=255)

    class PrivacyMeta:
        def export(self, instance):
            return {
```

(continues on next page)

(continued from previous page)

```
                'name': instance.name,
                'lower': instance.lower(),
            }
```

CHAPTER 4

Anonymising objects

4.1 Models

Call this to anonymise the private fields on the object.

4.1.1 `obj.anonymise()`

Once an object is anonymised a reference to that anonymisation will be recorded in `PrivacyAnonymised`.

4.1.2 `obj.is_anonymised = BooleanField()`

This is a boolean value stored in the database to register whether the object has been anonymised or not.

4.2 How anonymisation works

If a field is nullable, the value will be set to `None` (or in the case of blankable strings, `''`).

If a field is not nullable, the value will be set to a sensible default:

- Numbers will be set to `0`
- Strings will be set to a string representation of the primary key field
- Booleans will be set to `False` (although `BooleanField(null=True)` will always be nullable)
- `DateField` and `DateTimeField` will be set to the current date and time
- `TimeField` will be set to `00:00`
- `DurationField` will be set to `timedelta(0)`
- `EmailField` will be anonymised to `{pk}@anon.example.com`

- `URLField` will be anonymised to `{pk}@anon.example.com`
- `GenericIPAddressField` will be set to `0.0.0.0`
- `UUIDField` will be set to `{00000000-0000-0000-0000-000000000000}`

These default actions can be overridden by defining a custom anonymiser as `anonymise_<field_name>` method on the `PrivacyMeta` class - see the *PrivacyMeta* documentation for more details.

Custom field types will also need a custom anonymiser to be defined.

Some fields cannot be anonymised unless they can be null, and trying to anonymise them without a custom anonymiser will raise a `gdpr_assist.AnonymiseError` exception:

- File fields (`FilePathField`, `FileField`, `ImageField`)
- Relationships (`OneToOneField`, `ForeignKey`)

To ensure data integrity, trying to anonymise a `ManyToManyField` will always raise a `gdpr_assist.AnonymiseError`, unless you are using a custom anonymiser for that field.

The anonymiser cannot anonymise the primary key.

4.3 Anonymising related objects

When anonymising an object, its related objects will not be anonymised automatically. This is to prevent unintentional side-effects.

4.3.1 Using signals

To cascade an anonymisation to another model, use signals:

`gdpr_assist.signals.pre_anonymise`

Called before the object is anonymised. You will be passed the original object.

Example:

```
from gdpr_assist.signals import pre_anonymise

@receiver(pre_anonymise, sender=MyModel)
def anonymise_related(sender, instance, **kwargs):
    instance.my_related_obj.anonymise()
```

`gdpr_assist.signals.post_anonymise`

Called after anonymisation of the object. You will be passed the anonymised object.

Example:

```
from gdpr_assist.signals import post_anonymise

@receiver(post_anonymise, sender=MyModel)
def anonymise_related(sender, instance, **kwargs):
    instance.my_related_obj.anonymise()
```

4.3.2 When deleting an object

When an object is deleted, any objects related to the object as `on_delete=gdpr_assist.ANONYMISE(..)` will be anonymised. This takes one argument - the type of `on_delete` to perform on the field itself; for example:

```
class MyModel(models.Model):
    fk = models.ForeignKey(
        TargetModel,
        on_delete=gdpr_assist.ANONYMISE(models.SET_NULL),
    )
```

`ANONYMISE(..)` cannot take `CASCADE` or `PROTECTED` as arguments.

Deleting querysets

gdpr-assist modifies the queryset of registered models so that a bulk deletion will anonymise any related objects which use `ANONYMISE(..)`.

Note on `use_in_migrations` usage. If the model registered's objects manager sets use_in_migrations=‘‘use_in_migrations = True‘‘ the default queryset (`objects`) will not be changed, it will instead be available at the name set on register(..., gdpr_default_manager_name=”abc”) this is to allow for registering of third party models which make use of use_in_migrations.

Note that Django does not send delete signals for bulk delete operations in other for situations, so to anonymise related objects when a queryset is deleted, make sure the model being deleted is registered with gdpr-assist.

CHAPTER 5

Commands

5.1 Re-running deletions and anonymisations

To re-run a set of deletions and anonymisations, make sure your log database is available, then run:

```
./manage.py gdpr_rerun
```

5.2 Anonymising all personal data

To anonymise all data in all models registered with gdpr-assist:

```
./manage.py anonymise_db
```

This will anonymise all data in the database,

This command can be useful when working on a stage or local copy of the live database. Because it is probably a bad idea to run this on a production database, you will need to enable this command with the setting `GDPR_CAN_ANONYMISE_DATABASE = True`.

CHAPTER 6

The admin site

6.1 Bulk anonymisation

To add an "Anonymise" option to the actions list for a ModelAdmin, subclass gdpr_assist.admin.ModelAdmin:

```
import gdpr_assist
class MyAdmin(gdpr_assist.admin.ModelAdmin):
    ...
admin.site.register(MyModel, MyAdmin)
```

6.2 Personal data tool

In the admin site, under GDPR, select Personal data. This is a tool which lets you find, export, delete and anonymise personal data.

Submitting the search will call the PrivacyMeta.search() method on all models registered with gdpr-assist.

From there, records can be selected for export, anonymisation or deletion.

This tool is only available to superusers.

CHAPTER 7

Upgrading

For an overview of what has changed between versions, see the *Changelog*.

7.1 Instructions

7.1.1 Upgrading from 1.1.0

Anonymisation flag

Version 1.2.0 changes the way the anonymisation flag is stored. Previously it was stored in an `anonymised` field which gdpr-assist added to your models, but this caused problems when wanting to anonymise third party models. This flag has now been moved to a new model in the gdpr-assist app, linked to your objects using a generic foreign key.

If you migrate without following these instructions, you will lose information about which database objects have been anonymised.

Migrating your data

You must write a data migration to move this data to run before you create a migration to remove the `anonymised` field. There is a migration operator to help you:

1. Create empty migrations for your apps with existing anonymisable models:

   ```
   ./manage.py makemigrations myapp --empty
   ```

2. Add the operator using the following migration template:

   ```
   from django.db.migrations import Migration
   from gdpr_assist.upgrading import MigrateGdprAnonymised
   
   ```

(continues on next page)

(continued from previous page)

```
class Migration(migrations.Migration):
    dependencies = [
        ('myapp', '0012_migration'),  # Added by makemigrations
        ('gdpr_assist', '0002_privacyanonymised'),  # Keep this dependency
    ]
    operations = [
        MigrateGdprAnonymised('MyModelOne'),  # Update this to your model
        MigrateGdprAnonymised('MyModelSix'),  # Repeat for all your GDPR models
    ]
```

3. Create migrations to remove the fields:

```
./manage.py makemigrations myapp
```

4. Repeat for any other apps with anonymisable models
5. Run all migrations

   ./manage.py migrate

   ./manage.py migrate –database=gdpr_log

System check gdpr_assist.E001

Version 1.2.0 onwards adds a system check to ensure you have followed the above instructions, to avoid accidental data loss when upgrading. If your migration tries to remove the field before you have migrated data, you will see the error message:

```
Removing anonymised field before its data is migrated
```

This is triggered when removing any field called `anonymised` before it has been migrated with the `MigrateGdprAnonymised` operator.

In most cases you can fix this by following the instructions above.

If the `anonymised` field was not added by gdpr-assist, and you do not want to run `MigrateGdprAnonymised`, you can tell the check to ignore the failing migration by adding `gdpr_assist_safe = True` to the migration class; for example:

```
class Migration(migrations.Migration):
    gdpr_assist_safe = True
    dependencies = [...
```

Alternatively if you are happy that all your migrations are safe, you can add the check to `SILENCED_SYSTEM_CHECKS` in your project settings to disable the migration check:

```
SILENCED_SYSTEM_CHECKS = [
    "gdpr_assist.E001",
]
```

Changes to your code

In most cases no further action will be required, but if you are using the `anonymised` field in your own code, you will need to call `is_anonymised()` or query the model `gdpr_assist.models.PrivacyAnonymised` instead.

7.2 Changelog

7.2.1 1.4.2, 2022-04-28

Fix admin styling issue on person search.

7.2.2 1.4.1, 2022-02-25

Fix migration issue caused in supporting 2.2/without default_auto_field. Thanks @mserrano07 @llexical.

7.2.3 1.4.0, 2022-01-19

Features:

- Add support for Django 3.2, 4.0.
- Updated example project.
- Improve performance of log database and added `GDPR_LOG_ON_ANONYMISE` option to disable logging.

Fix:

- Resolve issue 48, use_in_migrations. Managers with use_in_migrations=True will no longer be cast, instead a

duplicate is created using the name provided at register(. . . , gdpr_default_manager_name="abc"). * Update PrivacyModel cast to support inheriting from another privacy model. * Performance improvements to for management command thanks @jayaddison-collabora

7.2.4 1.3.0, 2020-08-14

Features:

- Add support for Django 3.0 and 3.1

Changes:

- Remove support for Django 2.1 and earlier

7.2.5 1.2.0, 2020-07-15

Features:

- Add support for Django 2.2
- Add `can_anonymise` flag to `PrivacyMeta` to support searching and exporting data which shouldn't be anonymised. (#15, #17)
- Add bulk anonymisation operations to improve efficiency of large anonymisations

Changes:

- Remove support for Django 1.8

Bugfixes:

- Fix support for third party models by removing the `anonymised` field (#5, #13)
- Fix duplicate migrations (#6, #12)

- Fix documentation for post_anonymise (#8, #14)

Internal:

- Code style updated to use black and isort

7.2.6 1.1.0, 2020-03-20

Bugfix:

- Allow managers with delete to have custom additional parameters.

Other:

- This version removes python 2.7 support.

7.2.7 1.0.1, 2018-10-23

Bugfix:

- Managers on registered models which set `use_in_migrations` can now be serialised for migrations.

7.2.8 1.0.0, 2018-09-16

Initial public release

CHAPTER 8

Contributing

Contributions are welcome by pull request. Check the github issues and project *roadmap* to see what needs work.

8.1 Installing

The easiest way to work on GDPR-assist is to fork the project on github, then install it to a virtualenv:

```
virtualenv django-gdpr-assist
cd django-gdpr-assist
source bin/activate
pip install -e git+git@github.com:USERNAME/django-gdpr-assist.git#egg=django-gdpr-
↪assist[dev]
```

(replacing `USERNAME` with your username).

This will install the development dependencies too, and you'll find the source ready for you to work on in the `src` folder of your virtualenv.

8.2 Testing

Contributions will be merged more quickly if they are provided with unit tests.

Use `setup.py` to run the python tests on your current python environment; you can optionally specify which test to run:

```
python setup.py test [tests[.test_set.TestClass]]
```

Use `tox` to run them on one or more supported versions:

```
tox [-e py36-django1.11] [tests[.test_module.TestClass]]
```

Tox will also generate a `coverage` HTML report.

You can also use `detox` to run the tests concurrently, although you will need to run `tox -e report` again afterwards to generate the coverage report.

To use a different database (mysql, postgres etc) use the environment variables `DATABASE_ENGINE`, `DATABASE_NAME`, `DATABASE_USER`, `DATABASE_PASSWORD`, `DATABASE_HOST` and `DATABASE_PORT`, eg:

```
DATABASE_ENGINE=pgsql DATABASE_NAME=gdpr_assist_test [...] tox
```

8.3 Code overview

The `handlers.register_model` handler watches for new model definitions which include a `PrivacyMeta` attribute. These models are then registered automatically with the `registry.registry`.

Registration casts and instantiates the `PrivacyMeta` and stores it on the `_privacy_meta` attribute of the model. It also changes the base class of the model to `models.PrivacyModel`, its manager to `models.PrivacyManager` and its queryset to `models.PrivacyQuerySet` to add the necessary anonymisation attributes and methods.

Note on `use_in_migrations` usage. If the model registered's objects manager sets use_in_migrations=``use_in_migrations = True`` *objects* is not cast, instead, `gdpr_default_manager_name` must be used to give an alternate name.

`Model.anonymisable_manager()` can also be used to access the PrivacyManager regardless of `gdpr_default_manager_name`.

Once all models are registered, `apps.GdprAppConfig.ready` looks at all registered models for a `OneToOneField` or `ForeignKey` which have `on_delete=ANONYMISE(..)`, and then logs the related models with the registry so that `handlers.handle_pre_delete` knows to watch them.

When a registered object is deleted, its details are logged to `models.EventLog`, stored in a separate database.

Anonymisation starts with `models.PrivacyModel.anonymise`, which then calls the field-specific anonymise functions in the `PrivacyMeta` instance; fields which do not have one defined use `anonymiser.anonymise_field`.

8.4 Known limitations

- QuerySet bulk deletions on a model will not be detected unless it has a `PrivacyMeta` or is manually registered with `gdpr_assist.register`
- Operations involving gdpr-assist may be slower than normal (ie bulk deletions) due to the additional processing required.

8.5 Roadmap

Features planned for future releases:

- Settings to customise the `anonymise()` method name on registered models - see *Anonymising objects*
- Subclass the queryset of `on_delete=ANONYMISE(..)` related models which aren't registered, so that bulk deletion always results in anonymisation - see *Anonymising objects*
- Ability to change a relationship field on a registered third-party model to use `on_delete=ANONYMISE(..)`

- A generic view to allow self-service data export, ready to be added to user-facing profile management.
- A generic view to allow self-service data removal, ready to be added to user-facing profile management.

This app does not currently attempt to provide any sort of framework for managing opt-in or consent, because in our experience no two sites are similar enough for a generic solution.