djfernet Documentation

Release 0.5.dev1

Carl Meyer

Feb 13, 2022

CONTENTS

CHAPTER

ONE

DJFERNET

Maintained fork of github.com/orcasgit/django-fernet-fields used by yourlabs.io/oss/djwebdapp

Fernet symmetric encryption for Django model fields, using the cryptography library.

`djfernet` supports Django 4.0 and later on Python 3.

Only PostgreSQL, SQLite, and MySQL are tested, but any Django database backend with support for `BinaryField` should work.

Danger: If you have data created with djfernet < 0.8, you will need the following setting to be able to decrypt existing data: `DJFERNET_PREFIX = b'djfernet'`. Make sure you use a bytestring right there with `b'..'`!!

1.1 Getting Help

Documentation for djfernet is available at https://djfernet.readthedocs.org/

This app is available on PyPI and can be installed with `pip install djfernet`.

1.2 Prerequisites

Only PostgreSQL, SQLite, and MySQL are tested, but any Django database backend with support for `BinaryField` should work.

1.3 Installation

`djfernet` is available on PyPI. Install it with:

```
pip install djfernet
```

1.4 Usage

Just import and use the included field classes in your models:

```
from django.db import models
from fernet_fields import EncryptedTextField


class MyModel(models.Model):
    name = EncryptedTextField()
```

You can assign values to and read values from the `name` field as usual, but the values will automatically be encrypted before being sent to the database and decrypted when read from the database.

Encryption and decryption are performed in your app; the secret key is never sent to the database server. The database sees only the encrypted value of this field.

1.4.1 Field types

Several other field classes are included: `EncryptedCharField`, `EncryptedEmailField`, `EncryptedIntegerField`, `EncryptedDateField`, and `EncryptedDateTimeField`. All field classes accept the same arguments as their non-encrypted versions.

To create an encrypted version of some other custom field class, inherit from both `EncryptedField` and the other field class:

```
from fernet_fields import EncryptedField
from somewhere import MyField

class MyEncryptedField(EncryptedField, MyField):
    pass
```

1.4.2 Nullable fields

Nullable encrypted fields are allowed; a `None` value in Python is translated to a real `NULL` in the database column. Note that this trivially reveals the presence or absence of data in the column to an attacker. If this is a problem for your case, avoid using a nullable encrypted field; instead store some other sentinel "empty" value (which will be encrypted just like any other value) in a non-nullable encrypted field.

1.5 Keys

By default, `djfernet` uses your `SECRET_KEY` setting as the encryption key.

You can instead provide a list of keys in the `FERNET_KEYS` setting; the first key will be used to encrypt all new data, and decryption of existing values will be attempted with all given keys in order. This is useful for key rotation: place a new key at the head of the list for use with all new or changed data, but existing values encrypted with old keys will still be accessible:

```
FERNET_KEYS = [
    'new key for encrypting',
```

(continues on next page)

(continued from previous page)

```
    'older key for decrypting old data',
]
```

Warning: Once you start saving data using a given encryption key (whether your `SECRET_KEY` or another key), don't lose track of that key or you will lose access to all data encrypted using it! And keep the key secret; anyone who gets ahold of it will have access to all your encrypted data.

1.5.1 Disabling HKDF

Fernet encryption requires a 32-bit url-safe base-64 encoded secret key. By default, `djfernet` uses HKDF to derive such a key from whatever arbitrary secret key you provide.

If you wish to disable HKDF and provide your own Fernet-compatible 32-bit key(s) (e.g. generated with Fernet.generate_key()) directly instead, just set `FERNET_USE_HKDF = False` in your settings file. If this is set, all keys specified in the `FERNET_KEYS` setting must be 32-bit and url-safe base64-encoded bytestrings. If a key is not in the correct format, you'll likely get "incorrect padding" errors.

Warning: If you don't define a `FERNET_KEYS` setting, your `SECRET_KEY` setting is the fallback key. If you disable HKDF, this means that your `SECRET_KEY` itself needs to be a Fernet-compatible key.

1.5.2 Indexes, constraints, and lookups

Because Fernet encryption is not deterministic (the same source text encrypted using the same key will result in a different encrypted token each time), indexing or enforcing uniqueness or performing lookups against encrypted data is useless. Every encrypted value will always be different, and every exact-match lookup will fail; other lookups' results would be meaningless.

For this reason, `EncryptedField` will raise `django.core.exceptions.ImproperlyConfigured` if passed any of `db_index=True`, `unique=True`, or `primary_key=True`, and any type of lookup on an `EncryptedField` except for `isnull` will raise `django.core.exceptions.FieldError`.

1.5.3 Ordering

Ordering a queryset by an `EncryptedField` will not raise an error, but it will order according to the encrypted data, not the decrypted value, which is not very useful and probably not desired.

Raising an error would be better, but there's no mechanism in Django for a field class to declare that it doesn't support ordering. It could be done easily enough with a custom queryset and model manager that overrides `order_by()` to check the supplied field names. You might consider doing this for your models, if you're concerned that you might accidentally order by an `EncryptedField` and get junk ordering without noticing.

1.5.4 Migrations

If migrating an existing non-encrypted field to its encrypted counterpart, you won't be able to use a simple `AlterField` operation. Since your database has no access to the encryption key, it can't update the column values correctly. Instead, you'll need to do a three-step migration dance:

1. Add the new encrypted field with a different name and initialize its values as *null*, otherwise decryption will be attempted before anything has been encrypted.
2. Write a data migration (using RunPython and the ORM, not raw SQL) to copy the values from the old field to the new (which automatically encrypts them in the process).
3. Remove the old field and (if needed) rename the new encrypted field to the old field's name.

1.6 Contributing

Thanks for your interest in contributing! The advice below will help you get your issue fixed / pull request merged.

Please file bugs and send pull requests to the GitHub repository and issue tracker.

1.6.1 Submitting Issues

Issues are easier to reproduce/resolve when they have:

- A pull request with a failing test demonstrating the issue
- A code example that produces the issue consistently
- A traceback (when applicable)

1.6.2 Pull Requests

When creating a pull request:

- Write tests (see below)
- Note user-facing changes in the CHANGES file
- Update the documentation as needed
- Add yourself to the AUTHORS file

1.6.3 Testing

Please add tests for any changes you submit. The tests should fail before your code changes, and pass with your changes. Existing tests should not break. Coverage (see below) should remain at 100% following a full tox run.

To install all the requirements for running the tests:

```
pip install -r requirements.txt
```

The tests also require that you have a local PostgreSQL server running and a user with create-database permissions. The tests will use a database named `djftest`; if it already exists it will be wiped and re-created.

To run the tests once:

```
./runtests.py
```

To run tox (which runs the tests across all supported Python and Django versions) and generate a coverage report in the `htmlcov/` directory:

```
make test
```

This requires that you have `python2.7`, `python3.3`, `python3.4`, `pypy`, and `pypy3` binaries on your system's shell path.

To install PostgreSQL on Debian-based systems:

```
$ sudo apt-get install postgresql
```

You'll need to run the tests as a user with permission to create databases. By default, the tests attempt to connect as a user with your shell username. You can override this by setting the environment variable `DJF_USERNAME`.

1.7 Changelog

1.7.1 0.8.0

Switched back to django-fernet-fields for default salt, making it incompatible with 0.7.4! But, compatible with django-fernet-fields, so that you can migrate easily if you haven't already.

If, unfortunnately, you have already deployed this in production, you have two options:

- re-encrypt your data to use django-fernet-fields instead of djfernet,
- or set settings.DJFERNET_PREFIX=djfernet to keep going

Sorry about this laborious releases.

Also, added EncryptedBinaryField.

1.7.2 0.7.4

First release since fork. Unfortunnately, my sed s/django-fernet-fields/djfernet/ caused a change in the salt, make it impossible to decrypt existing data!!

Added settings.DJFERNET_PREFIX to set it to django-fernet-fields and make it compatible again through a setting.

Thanks to @sevdog for the report.

1.7.3 0.6 (2019.05.10)

- Support Postgres 10
- Drop support for Django < 1.11, Python 3.3/3.4
- Add support for Django 1.11 through 2.2, Python 3.7

1.7.4 0.5 (2017.02.22)

- Support Django 1.10.

1.7.5 0.4 (2015.09.18)

- Add *isnull* lookup.

1.7.6 0.3 (2015.05.29)

- Remove DualField and HashField. The only cases where they are useful, they aren't secure.

1.7.7 0.2.1 (2015.05.28)

- Fix issue getting IntegerField validators.

1.7.8 0.2 (2015.05.28)

- Extract HashField for advanced lookup needs.

1.7.9 0.1 (2015.05.27)

- Initial working version.