django-cryptography Documentation

Release 1.0

George Marshall

Feb 10, 2020

Contents

A set of primitives for easily encrypting data in Django, wrapping the Python Cryptography library. Also provided is a drop in replacement for Django's own cryptographic primitives, using Cryptography as the backend provider.

CHAPTER 1

Why another encryption library for Django?

The motivation for making django-cryptography was from the general frustration of existing solutions. Libraries such as django-cryptographic-fields and django-crypto-fields do not allow a way to easily work with custom fields, being limited to their own provided subset. As well as many others lacking Python 3 and modern Django support.

1.1 Installation

1.1.1 Requirements

- Python (3.5, 3.6, 3.7, 3.8)
- Cryptography (2.0+)
- Django (1.11, 2.2, 3.0)

```
pip install django-cryptography
```

1.2 Settings

1.2.1 CRYPTOGRAPHY_BACKEND

Default: `cryptography.hazmat.backends.default_backend()`

1.2.2 CRYPTOGRAPHY_DIGEST

Default: `cryptography.hazmat.primitives.hashes.SHA256`

The digest algorithm to use for signing and key generation.

1.2.3 CRYPTOGRAPHY_KEY

Default: `None`

When value is `None` a key will be derived from `SECRET_KEY`. Otherwise the value will be used for the key.

1.2.4 CRYPTOGRAPHY_SALT

Default: `'django-cryptography'`

1.2.5 Drop-in Replacements

SIGNING_BACKEND

The default can be replaced with a a Cryptography based version.

```
SIGNING_BACKEND = 'django_cryptography.core.signing.TimestampSigner'
```

1.3 Fields

django_cryptography.fields.**encrypt**(*base_field*, *key=None*, *ttl=None*)

A decorator for creating encrypted model fields.

Parameters

- **key** (*bytes*) – This is an optional argument.

  Allows for specifying an instance specific encryption key.

- **ttl** (*int*) – This is an optional argument.

  The amount of time in seconds that a value can be stored for. If the time to live of the data has passed, it will become unreadable. The expired value will return an *Expired* object.

Return type models.Field[*EncryptedMixin*, T]

class django_cryptography.fields.**PickledField**(**args*, ***kwargs*)

A field for storing pickled objects

deconstruct()

Return enough information to recreate the field as a 4-tuple:

- The name of the field on the model, if contribute_to_class() has been run.
- The import path of the field, including the class:e.g. django.db.models.IntegerField This should be the most portable version, so less specific may be better.
- A list of positional arguments.
- A dict of keyword arguments.

Note that the positional or keyword arguments must contain values of the following types (including inner values of collection types):

- None, bool, str, int, float, complex, set, frozenset, list, tuple, dict
- UUID
- datetime.datetime (naive), datetime.date

- top-level classes, top-level functions - will be referenced by their full import path
- Storage instances - these have their own deconstruct() method

This is because the values here must be serialized into a text format (possibly new Python code, possibly JSON) and these are the only types with encoding handlers defined.

There's no need to return the exact way the field was instantiated this time, just ensure that the resulting field is the same - prefer keyword arguments over positional ones, and omit parameters with their default values.

get_db_prep_value(*value*, *connection*, *prepared=False*)
Return field's value prepared for interacting with the database backend.

Used by the default implementations of get_db_prep_save().

get_default()
Return the default value for this field.

to_python(*value*)
Convert the input value into the expected Python data type, raising django.core.exceptions.ValidationError if the data can't be converted. Return the converted value. Subclasses should override this.

value_to_string(*obj*)
Pickled data is serialized as base64

1.3.1 Constants

django_cryptography.fields.**Expired = <object object>**
Represents an expired encryption value.

1.3.2 Helpers

django_cryptography.fields.**get_encrypted_field**(*base_class*)
A get or create method for encrypted fields, we cache the field in the module to avoid recreation. This also allows us to always return the same class reference for a field.

Return type models.Field[*EncryptedMixin*, T]

class django_cryptography.fields.**EncryptedMixin**(**args*, ***kwargs*)
A field mixin storing encrypted data

Parameters

- **key** (*bytes*) – This is an optional argument.

  Allows for specifying an instance specific encryption key.

- **ttl** (*int*) – This is an optional argument.

  The amount of time in seconds that a value can be stored for. If the time to live of the data has passed, it will become unreadable. The expired value will return an *Expired* object.

get_db_prep_save(*value*, *connection*)
Return field's value prepared for saving into a database.

1.4 Migrating existing data

See also:

If you are unfamiliar with migrations in Django, please consult the Django Migrations documentation.

To migrate an unencrypted database field to an encrypted field the following steps must be followed. Each step is labeled with its Django migration type of schema or data.

1. Rename existing field using a prefix such as `old_` (schema)
2. Add new encrypted field with name of the original field (schema)
3. Copy data from the old field into the new field (data)
4. Remove the old field (schema)

The steps are illustrated bellow for the following model:

```
class EncryptedCharModel(models.Model):
    field = encrypt(models.CharField(max_length=15))
```

Create the initial migration for the *EncryptedCharModel*.

```
class Migration(migrations.Migration):

    initial = True

    dependencies = []

    operations = [
        migrations.CreateModel(
            name='EncryptedCharModel',
            fields=[
                ('id', models.AutoField(
                    auto_created=True,
                    primary_key=True,
                    serialize=False,
                    verbose_name='ID')),
                ('field', models.CharField(max_length=15)),
            ],
        ),
    ]
```

Rename the old field by pre-fixing as `old_field` from `field`

```
class Migration(migrations.Migration):

    dependencies = [
        ('fields', '0001_initial'),
    ]

    operations = [
        migrations.RenameField(
            model_name='encryptedcharmodel',
            old_name='field',
            new_name='old_field',
        ),
    ]
```

Add the new encrypted field using the original name from our field.

```
class Migration(migrations.Migration):

    dependencies = [
        ('fields', '0002_rename_fields'),
    ]

    operations = [
        migrations.AddField(
            model_name='encryptedcharmodel',
            name='field',
            field=django_cryptography.fields.encrypt(
                models.CharField(default=None, max_length=15)),
            preserve_default=False,
        ),
    ]
```

Copy the data from the old field into the new field using the ORM. Providing forwards and reverse methods will allow restoring the field to its unencrypted form.

```
def forwards_encrypted_char(apps, schema_editor):
    EncryptedCharModel = apps.get_model("fields", "EncryptedCharModel")

    for row in EncryptedCharModel.objects.all():
        row.field = row.old_field
        row.save(update_fields=["field"])


def reverse_encrypted_char(apps, schema_editor):
    EncryptedCharModel = apps.get_model("fields", "EncryptedCharModel")

    for row in EncryptedCharModel.objects.all():
        row.old_field = row.field
        row.save(update_fields=["old_field"])


class Migration(migrations.Migration):

    dependencies = [
        ("fields", "0003_add_encrypted_fields"),
    ]

    operations = [
        migrations.RunPython(forwards_encrypted_char, reverse_encrypted_char),
    ]
```

Delete the old field now that the data has been copied into the new field

```
class Migration(migrations.Migration):

    dependencies = [
        ('fields', '0004_migrate_data'),
    ]

    operations = [
        migrations.RemoveField(
            model_name='encryptedcharmodel',
```

(continues on next page)

(continued from previous page)

```
            name='old_field',
        ),
    ]
```

1.5 Cryptography by example

Using symmetrical encryption to store sensitive data in the database. Wrap the desired model field with *encrypt()* to easily protect its contents.

```
from django.db import models

from django_cryptography.fields import encrypt


class MyModel(models.Model):
    name = models.CharField(max_length=50)
    sensitive_data = encrypt(models.CharField(max_length=50))
```

The data will now be automatically encrypted when saved to the database. *encrypt()* uses an encryption that allows for bi-directional data retrieval.

1.6 Releases

1.6.1 1.0 - 2020-02-09

- Added support Django 3.0
- Dropped Django 2.1 support
- Dropped Python 2.7 support
- Removed legacy support code

1.6.2 0.4 - 2020-01-28

- Dropped Django 1.8 and 2.0 support
- Fixed Django 3.0 deprecation warning
- Fixed migration test cases

1.6.3 0.3 - 2017-12-19

- Fixed issue with Django migration generation
- Added initial support for Django 2.0
- Dropped Python 3.3 support

1.6.4 0.2 - 2016-12-06

- Refactored `EncryptedField` into *`encrypt()`* decorator.

1.6.5 0.1 - 2016-05-21

- Initial release

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index