django-anysign Documentation

Release 0.4.dev0

Benoît Bryon

June 18, 2015

Contents

django-anysign is a Django application to manage online signature in a generic way.

Its goal is to provide a consistent API whatever the signature implementation. So that, using *django-anysign*, you can easily switch from one signature backend to another, or use several backends at once.

CHAPTER 1

Project status

django-anysign is under active development. The project is not mature yet, but authors already use it! It means that, while API and implementation may change (improve!) a bit, authors do care of the changes.

Also, help is welcome! Feel free to report issues, request features or refactoring!

CHAPTER 2

Resources

- Documentation: https://django-anysign.readthedocs.org
- Bugtracker: https://github.com/novafloss/django-anysign/issues
- Changelog: https://django-anysign.readthedocs.org/en/latest/about/changelog.html
- Roadmap: https://github.com/novafloss/django-anysign/milestones
- Code repository: https://github.com/novafloss/django-anysign
- Continuous integration: https://travis-ci.org/novafloss/django-anysign

CHAPTER 3

Contents

3.1 Install

django-anysign is open-source software, published under BSD license. See License for details.

If you want to install a development environment, you should go to Contributing documentation.

3.1.1 Prerequisites

- Python [1] 2.7, 3.3 or 3.4. Other versions may work, but they are not part of the test suite at the moment.

3.1.2 As a library

In most cases, you will use *django-anysign* as a dependency of another project. In such a case, you should add `django-anysign` in your main project's requirements. Typically in `setup.py`:

```
from setuptools import setup

setup(
    install_requires=[
        'django-anysign',
        #...
    ]
    # ...
)
```

Then when you install your main project with your favorite package manager (like pip [2]), *django-anysign* will automatically be installed.

3.1.3 Standalone

You can install *django-anysign* with your favorite Python package manager. As an example with pip [2]:

```
pip install django-anysign
```

[1] https://www.python.org/
[2] https://pypi.python.org/pypi/pip/

3.1.4 Check

Check *django-anysign* has been installed:

```
python -c "import django_anysign;print(django_anysign.__version__)"
```

You should get *django_anysign*'s version.

References

3.2 Configure

Here is the list of settings used by *django-anysign*.

3.2.1 INSTALLED_APPS

There is no need to register *django-anysign* application in your Django's `INSTALLED_APPS` setting.

3.2.2 ANYSIGN

The `settings.ANYSIGN` is a dictionary that contains all specific configuration for *django-anysign*.

Example from the Demo project:

```
ANYSIGN = {
    'BACKENDS': {
        'dummysign': 'django_dummysign.backend.DummySignBackend',
    },
    'SIGNATURE_TYPE_MODEL': 'django_anysign_demo.models.SignatureType',
    'SIGNATURE_MODEL': 'django_anysign_demo.models.Signature',
    'SIGNER_MODEL': 'django_anysign_demo.models.Signer',
}
```

BACKENDS

A dictionary where:

- keys are backend codes, i.e. machine-readable names for backends. These keys are typically stored in the database as *`django_anysign.models.SignatureType.signature_backend_code`*.
- values are Python path to import backend's implementation, typically a class.

See also *`get_signature_backend()`*.

SIGNATURE_TYPE_MODEL

The Python path to import the *SignatureType* model.

SIGNATURE_MODEL

The Python path to import the *Signature* model.

SIGNER_MODEL

The Python path to import the *Signer* model.

3.3 Models

django-anysign presumes digital signature involves models in the Django project: one to store the signatures, another to store signers, and one to store backend specific options.

That said, *django-anysign* does not embeds concrete models: it provides base models you have to extend in your applications. This design allows you to customize models the way you like, i.e. depending on your use case.

3.3.1 Minimal integration

Here is the minimal integration you need in some `models.py`:

```
import django_anysign


class SignatureType(django_anysign.SignatureType):
    pass


class Signature(django_anysign.SignatureFactory(SignatureType)):
    pass


class Signer(django_anysign.SignerFactory(Signature)):
    pass
```

The example above is taken from *django-anysign*'s Demo project.

3.3.2 SignatureType

class `django_anysign.models.`**`SignatureType`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Abstract base model for signature type.

A signature type encapsulates backend setup. Typically:

- a "configured backend" is a backend class (such as :class:~'django-dummysign.backend.DummySignBackend') and related configuration (URL, credentials...).
- a `Signature` instance will be related to a configured backend, via a `SignatureType`.

`signature_backend_code` = **None**

Machine-readable code for the backend. Typically related to settings, by default keys in `settings.ANYSIGN['BACKENDS']` dictionary.

class `Meta`

`abstract` = **False**

SignatureType.**signature_backend_options**
Dictionary for backend's specific configuration.

Default implementation returns empty dictionary.

There are 2 main ways for you to setup backends with the right arguments:

- in the model subclassing this one, override this property. This is the good option if you can have several `SignatureType` instances for one backend, i.e. if `signature_backend_code` is not unique.
- in the backend's subclass, make `__init__()` read the Django settings or environment. This can be a good option if you have an unique `SignatureBackend` instance matching a backend (`signature_backend_code` is unique).

SignatureType.**get_signature_backend**()
Instanciate and return signature backend instance.

Default implementation uses `get_backend_instance()` with `signature_backend_code` as positional arguement and with `signature_backend_options()` as keyword arguments.

SignatureType.**signature_backend**
Return backend from internal cache or new instance.

If `signature_backend_code` changed since the last access, then the internal (instance level) cache is invalidated and a new instance is returned.

3.3.3 Signature

django_anysign.models.**SignatureFactory**(*SignatureType*)
Return base class for signature model, using `SignatureType` model.

This pattern is the best one we found at the moment to have an abstract base model `SignatureBase` with appropriate foreign key to `SignatureType` model. Feel free to propose a better option if you know one ;)

Here is what you get in the Demo project:

3.3.4 Signer

django_anysign.models.**SignerFactory**(*Signature*)
Return base class for signer model, using `Signature` model.

This pattern is the best one we found at the moment to have an abstract base model `Signer` with appropriate foreign key to `Signature` model. Feel free to propose a better option if you know one ;)

Here is what you get in the Demo project:

3.4 Backends

django-anysign's signature backend encapsulates signature workflow and integration with vendor specific implementation.

Note: The backend API is quite experimental. This document deals with both vision (concepts) and current implementation (which may improve).

3.4.1 Scope of a backend

A signature backend is typically known by models and views. They use the backend to perform vendor-specific operations. The backend contains vendor-specific implementation that has to be shared with several consumers such as models and views.

A signature backend also typically knowns the workflows. So it should be helpful for URL resolution.

3.4.2 django-anysign's SignatureBackend

Here is the current implementation of base backend.

class `django_anysign.backend.`**`SignatureBackend`**(*name*, *code*, *url_namespace='anysign'*, ***kwargs*)

Bases: `object`

Encapsulate signature workflow and integration with vendor backend.

Here is a typical workflow:

- *`SignatureType`* instance is created. It encapsulates the backend type and its configuration.
- A `Signature` instance is created. The signature instance has a signature type attribute, hence a backend.
- Signers are notified, by email, text or whatever. They get an hyperlink to the "signer view". The URL may vary depending on the signature backend.
- A signer goes to the backend's "signer view" entry point: typically a view that integrates backend specific form to sign a document.
- Most backends have a "notification view", for the third-party service to signal updates.
- Most backends have a "signer return view", where the signer is redirected when he ends the signature process (whatever signature status).
- The backend's specific workflow can be made of several views. At the beginning, there is a Signature instance which carries data (typically a document). At the end, Signature is done.

`name` = None
Human-readable name.

`code` = None
Machine-readable name. Should be lowercase alphanumeric only, i.e. PEP-8 compliant.

`url_namespace` = None
Namespace for URL resolution.

`send_signature`(*signature*)
Initiate the signature process.

At this state, the signature object has been configured.

Typical implementation consists in sending signer URL to first signer.

Raise `NotImplementedError` if the backend does not support such a feature.

`get_signer_url`(*signer*)
Return URL where signer signs document.

Raise `NotImplementedError` in case the backend does not support "signer view" feature.

Default implementation reverses `get_signer_url_name()` with `signer.pk` as argument.

get_signer_url_name()
Return URL name where signer signs document.

Raise `NotImplementedError` in case the backend does not support "signer view" feature.

Default implementation returns `anysign:signer`.

get_signer_return_url(*signer*)
Return absolute URL where signer is redirected after signing.

The URL must be **absolute** because it is typically used by external signature service: the signer uses external web UI to sign the document(s) and then the signature service redirects the signer to (this) *Django* website.

Raise `NotImplementedError` in case the backend does not support "signer return view" feature.

Default implementation reverses `get_signer_return_url_name()` with `signer.pk` as argument.

get_signer_return_url_name()
Return URL name where signer is redirected once document has been signed.

Raise `NotImplementedError` in case the backend does not support "signer return view" feature.

Default implementation returns `anysign:signer_return`.

get_signature_callback_url(*signature*)
Return URL where backend can post signature notifications.

Raise `NotImplementedError` in case the backend does not support "signature callback url" feature.

Default implementation reverses `get_signature_callback_url_name()` with `signature.pk` as argument.

get_signature_callback_url_name()
Return URL name where backend can post signature notifications.

Raise `NotImplementedError` in case the backend does not support "signer return view" feature.

Default implementation returns `anysign:signature_callback`.

create_signature(*signature*)
Register `signature` in backend, return updated object.

This method is typically called by views which create `Signature` instances.

If backend stores a signature object, then implementation should update `signature_backend_id`.

Base implementation does nothing: override this method in backends.

3.4.3 django-dummysign's SignatureBackend

Here is the demo signature backend implementation provided by *django-dummysign*.

```
import logging

import django_anysign


logger = logging.getLogger(__name__)


class DummySignBackend(django_anysign.SignatureBackend):
```

```
    def __init__(self):
        super(DummySignBackend, self).__init__(
            name='DummySign',
            code='dummysign',
        )

    def create_signature(self, signature):
        """Register ``signature`` in backend, return updated object.

        As a dummy backend: just emit a log.

        """
        signature = super(DummySignBackend, self).create_signature(signature)
        logger.debug('[django_dummysign] Signature created in backend')
        return signature
```

3.5 Views

At the moment, *django-anysign* does not provide views or generic views. But this feature is part of the Vision...

3.6 Loading

Since *django-anysign* does not provide concrete Models, and models are configured in settings, here are tools to load models and backends.

3.6.1 get_signature_backend

`django_anysign.loading.`**`get_signature_backend`**`(`*code*, **args*, ***kwargs*`)`

Instantiate instance for `backend_code` with `args` and `kwargs`.

Get the backend factory (class) using `settings.ANYSIGN['BACKENDS']`.

Positional and keyword arguments are proxied as is.

3.6.2 get_signature_type_model

`django_anysign.loading.`**`get_signature_type_model`**`()`

Return model defined as `settings.ANYSIGN['SIGNATURE_TYPE_MODEL']`.

3.6.3 get_signature_model

`django_anysign.loading.`**`get_signature_model`**`()`

Return model defined as `settings.ANYSIGN['SIGNATURE_MODEL']`.

3.6.4 get_signer_model

`django_anysign.loading.`**`get_signer_model`**`()`

Return model defined as `settings.ANYSIGN['SIGNER_MODEL']`.

3.7 Demo project

Demo folder in project's repository [3] contains a Django project to illustrate *django-anysign* usage. It basically integrates *django-dummysign* in a project.

Examples in the documentation are imported from the demo project.

Feel free to use the demo project as a sandbox. See Contributing for details about development environment setup.

Notes & references

3.8 About django-anysign

This section is about the *django-anysign* project itself.

3.8.1 Vision

django-anysign provides conventions and base resources to implement digital signature features within a Django project.

django-anysign's goal is to provide a consistent API whatever the signature implementation. This concept basically covers the following use cases:

- plug several signature backends and their specific workflows into a single website.
- in a website using a single signature backend, migrate from one backend to another with minimum efforts.
- as a developer, implement bindings for a new signature service vendor.

django-anysign presumes the following items are generally involved in digital signature features:

- models. Such as signature, signer and signature type (backend options).
- workflows. They usually start with the creation of a document to sign (setup a signature, assign signers, choose a backend). They usually end when the document has been signed by all signers. Steps between "start" and "end" typically vary depending on the vendor signature service.
- views. Most signature workflows use similar views, such as "create signature", "sign document", "signer processed document" or "API callback". Of course, the implementation and order vary depending on the vendor signature service. But some bits are generic.

django-anysign does not include vendor-specific implementation. Third-party projects do. And they can be based on *django-anysign*. So as a developer, you are likely to discover *django-anysign* via these vendor-specific projects. See Alternatives and related projects for details about third-party projects.

django-anysign is a framework. It does not provide all-in-one solutions. You may have to implement some things in your Django project. *django-anysign* tries to make this custom code easier to imagine and write, using conventions, utilities and base classes.

3.8.2 Alternatives and related projects

This document presents other projects that provide similar or complementary functionalities. It focuses on differences or relationships with *django-anysign*.

[3] https://github.com/novafloss/django-anysign/tree/master/demo/

django-dummysign

django-dummysign [4] provides a dummy backend that implements *django-anysign* API. It is made for tests, prototypes or developments.

Note: At the moment, *django-dummysign* is distributed as part of *django-anysign* itself. When you `pip install django-anysign` you get both `import django_anysign` and `django_dummysign`.

This happened because *django-anysign* and *django-dummysign* are developed together and tests from one require updates from the other, and vice-versa. They may be separated again later, as an example if *django-dummysign* gets additional requirements such as pyPdf you do not need in *django-anysign*.

---

django-docusign

django-docusign [5] provides a backend for DocuSign [6] signature service. It uses *django-anysign* to integrate pydocusign [7] in *Django*.

django-hello_sign

django-hello_sign [8] integrates hellosign [9] in *Django*. It does not use *django-anysign* API.

References

3.8.3 License

[4] https://github.com/novafloss/django-anysign/tree/master/django_dummysign/
[5] https://github.com/novafloss/django-docusign/
[6] https://www.docusign.com/
[7] https://github.com/novafloss/pydocusign/
[8] https://pypi.python.org/pypi/django-hello_sign/
[9] https://www.hellosign.com/

WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3.8.4 Authors & contributors

Maintainer: Benoît Bryon <benoit@marmelune.net>, as a member of the PeopleDoc [10] team: https://github.com/novapost/

Developers: https://github.com/novafloss/django-anysign/graphs/contributors

Notes & references

3.8.5 Changelog

This document describes changes between each past release. For information about future releases, check milestones [11] and Vision.

0.4 (unreleased)

Workaround identifiers. You will need schema and data migrations after the upgrade.

- Features #7 and #8 - `Signature` and `Signer` models have `anysign_internal_id` attribute. It is an unique identifier for signature or signer on Django side. For use as a "foreign key" in backend's database, whenever possible. It defaults to an UUID. You may override it with a custom property if your models already have some UUID.
- Feature #5 - `Signer` model has `signature_backend_id` attribute. Use it to store the backend's signer identifier, i.e. signer's identifier in external database.
- Refactoring #15 - Project repository moved to github.com/novafloss (was github.com/novapost).

0.3 (2014-10-08)

Signers' ordering.

- Feature #4 - Added `signing_order` attribute to `Signer` model.

0.2 (2014-09-12)

Minor fixes.

- Feature #2 - Explicitely mark Django 1.7 as not supported (tests fail with Django 1.7) in packaging.
- Bug #3 - Fixed wrong usage of *django-anysign* API in *django-dummysign* backend.

[10] http://www.people-doc.com
[11] https://github.com/novafloss/django-anysign/milestones

0.1 (2014-08-11)

Initial release.

- Introduced base model `SignatureType` and base model factories `SignatureFactory` and `SignerFactory`.
- Introduced base backend class `SignatureBackend`.
- Introduced loaders for custom models and backend: `get_signature_backend_instance`, `get_signature_type_model`, `get_signature_model` and `get_signer_model`.

Notes & references

3.9 Contributing

This document provides guidelines for people who want to contribute to the *django-anysign* project.

3.9.1 Create tickets

Please use django-anysign bugtracker [12] **before** starting some work:

- check if the bug or feature request has already been filed. It may have been answered too!
- else create a new ticket.
- if you plan to contribute, tell us, so that we are given an opportunity to give feedback as soon as possible.
- Then, in your commit messages, reference the ticket with some `refs #TICKET-ID` syntax.

3.9.2 Use topic branches

- Work in branches.
- Prefix your branch with the ticket ID corresponding to the issue. As an example, if you are working on ticket #23 which is about contribute documentation, name your branch like `23-contribute-doc`.
- If you work in a development branch and want to refresh it with changes from master, please rebase [13] or merge-based rebase [14], i.e. do not merge master.

3.9.3 Fork, clone

Clone *django-anysign* repository (adapt to use your own fork):

```
git clone git@github.com:novafloss/django-anysign.git
cd django-anysign/
```

[12] https://github.com/novafloss/django-anysign/issues
[13] http://git-scm.com/book/en/Git-Branching-Rebasing
[14] http://tech.novapost.fr/psycho-rebasing-en.html

3.9.4 Usual actions

The *Makefile* is the reference card for usual actions in development environment:

- Install development toolkit with pip [15]: `make develop`.
- Run tests with tox [16]: `make test`.
- Build documentation: `make documentation`. It builds Sphinx [17] documentation in *var/docs/html/index.html*.
- Release *django-anysign* project with zest.releaser [18]: `make release`.
- Cleanup local repository: `make clean`, `make distclean` and `make maintainer-clean`.

See also `make help`.

Notes & references

[15] https://pypi.python.org/pypi/pip/
[16] https://pypi.python.org/pypi/tox/
[17] https://pypi.python.org/pypi/Sphinx/
[18] https://pypi.python.org/pypi/zest.releaser/

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

A

C

G

N

S

U