konfetti Documentation

finance team

Jun 05, 2020

Contents:

1 Usage **1**
- 1.1 Lazy evaluation . . . 1
- 1.2 Environment . . . 1
- 1.3 Vault . . . 2
- 1.4 Testing . . . 6
- 1.5 Extending configuration . . . 8
- 1.6 Extras . . . 8
- 1.7 Django integration . . . 9
- 1.8 Flask extension . . . 9

2 Configuration 101 **11**
- 2.1 Do not access configuration on the module level . . . 11

3 Contributing **13**
- 3.1 Code formatting . . . 13
- 3.2 Testing . . . 13

4 Changelog **15**
- 4.1 Unreleased . . . 15
- 4.2 **'0.8.0'_** - 2020-01-15 . . . 15
- 4.3 0.7.2 - 2019-07-18 . . . 15
- 4.4 0.7.1 - 2019-07-17 . . . 15
- 4.5 0.7.0 - 2019-07-15 . . . 16
- 4.6 0.6.0 - 2019-06-30 . . . 16

5 Indices and tables **17**

CHAPTER 1

Usage

Table of contents:

- Lazy evaluation
- Environment
- Vault
- Testing

1.1 Lazy evaluation

Until a config option is accessed it is not evaluated - it is lazy. To avoid side effects on imports accessing configuration should be avoided on a module level.

This concept allows you to choose when you actually evaluate the config. Why?

- Testing. If you need to test a small piece of code that doesn't require any configuration - you don't have to setup it;
- Faster application startup; Use only what you need at the moment

It is still possible to evaluate the config eagerly on the app startup - access the needed variables in the entry points. It could be done either with direct accessing needed variables or with `config.require(...)` / `config.asdict()` calls.

1.2 Environment

```
from konfetti import env

VARIABLE = env("VARIABLE_NAME", default="foo")
```

Since environment variables are strings, there is a `cast` option to convert given variable from a string to the desired type:

```
from konfetti import env

VARIABLE = env("VARIABLE_NAME", default=42, cast=int)
```

You can pass any callable as `cast`. `list`, `tuple`, `set` and `frozenset` will recognize comma separated values:

```
LIST = env("LIST", cast=list)
# 1,2,3
In [1]: config.LIST
["1", "2", "3"]
```

Those container types accept also `subcast` that will be applied to each element of the container:

```
LIST = env("LIST", cast=list, subcast=int)
# 1,2,3
In [1]: config.LIST
[1, 2, 3]
```

If there is a need to use the environment variable immediately, it could be evaluated via `str` call (other ways could be added on demand):

```
from konfetti import env, vault

DATABASE_ROLE = env("DATABASE_ROLE", default="booking")

DATABASE_URI = vault(f"db/{DATABASE_ROLE}")
```

If `cast` is specified, then it will be applied before evaluation as well.

1.2.1 `.env` support

It is possible to specify a path to the `.env` file and it will be used as a source of data for environment variables.

`dotenv_override` parameter specifies whether the `.env` value should be used if both the environment variable and the `.env` record exists, `False` by default.

```
# app_name/settings/__init__.py
from konfetti import Konfig

config = Konfig(dotenv="path/to/.env", dotenv_override=False)
```

1.3 Vault

1.3.1 Backend configuration

To use Vault as a secrets storage you need to configure the access point:

```
# app_name/settings/__init__.py
from konfetti import Konfig, AsyncVaultBackend

config = Konfig(vault_backend=AsyncVaultBackend("your/prefix"))
```

There are two Vault backends available:

- `konfetti.VaultBackend`
- `konfetti.AsyncVaultBackend`

The main difference is that the latter requires using `await` to access the secret value (the call will be handled asynchronously under the hood), otherwise the interfaces and capabilities are the same.

Each backend requires a `prefix` to be specified, the trailing / leading slashes don't matter, `"your/prefix"` will work the same as `"/your/prefix/"`.

There are two ways to provide access credentials:

- via `VAULT_TOKEN` environment variable for token-based auth method
- via `VAULT_USERNAME` and `VAULT_PASSWORD` environment variables for `userpass` auth method

If both are provided, token will be tried first and `userpass` credentials next in case of expired token.

Access credentials must be specified as a part of configuration.

```
# app_name/settings/production.py
VAULT_TOKEN = env("VAULT_TOKEN")
VAULT_USERNAME = env("VAULT_USERNAME")
VAULT_PASSWORD = env("VAULT_PASSWORD")
```

1.3.2 Usage

Every Vault secret needs a `path` to be used as a lookup (leading and trailing slashes don't matter as well):

```
# app_name/settings/production.py
from konfetti import vault

WHOLE_SECRET = vault("path/to")
```

In this case all key/value pairs will be loaded on evaluation:

```
In [1]: from app_name.settings import config
In [2]: await config.WHOLE_SECRET
{'key': 'value', 'foo': 'bar'}
```

You can specify a specific key to be returned for a config option with `[]` syntax:

```
# app_name/settings/production.py
from konfetti import vault

KEY = vault("path/to")["key"]
```

```
In [1]: from app_name.settings import config
In [2]: await config.KEY
value
```

Using square brackets will not trigger evaluation - you could specify as many levels as you want:

```
# app_name/settings/production.py
from konfetti import vault

DEEP = vault("path/to")["deeply"]["nested"]["key"]
```

Casting could be specified as well:

```
# app_name/settings/production.py
from decimal import Decimal
from konfetti import vault

DECIMAL = vault("path/to", cast=Decimal)["fee_amount"]  # stored as string
```

```
In [1]: from app_name.settings import config
In [2]: await config.DECIMAL
Decimal("0.15")
```

Sometimes you need to access to some secrets dynamically. `Konfig` provides a way to do it:

```
In [1]: from app_name.settings import config
In [2]: await config.get_secret("path/to")["key"]
value
```

Secret files

It is possible to get a file-like interface for vault secret.

```
# app_name/settings/production.py
from konfetti import vault_file

KEY = vault_file("path/to/file")["key"]
```

```
In [1]: from app_name.settings import config
In [2]: (await config.KEY).readlines()
[b'value']
```

Defaults

It is possible to specify the default value for vault variable. Value could be any type for a key in a secret and a `dict` for the whole secret.

```
DEFAULT = vault("path/to", default="default")["DEFAULT"]
DEFAULT_SECRET = vault("path/to", default={"DEFAULT_SECRET": "default_secret"})

In [1]: from app_name.settings import config
In [2]: await config.DEFAULT
"default"
In [3]: await config.DEFAULT_SECRET
{"DEFAULT_SECRET": "default_secret"}
```

Defaults could be disabled entirely if `VAULT_DISABLE_DEFAULTS` is set

```
$ export VAULT_DISABLE_DEFAULTS="true"
```

Overriding Vault secrets

In some cases, secrets need to be overridden in runtime on the application level. You can define some custom values for tests or you just want to run the app with some different configuration without changing data in Vault.

There is a way to do it using environment variables or `.env` records To redefine certain config option you need to redefine the whole secret with a JSON encoded string.

Example:

```
# app_name/settings/production.py
from konfetti import vault

KEY = vault("path/to")["key"]
```

```
In [1]: from app_name.settings import config
In [2]: await config.KEY
value
In [3]: import os
In [4]: os.environ["PATH__TO"] = '{"key": "overridden"}'
In [5]: await config.KEY
overridden
```

To check how to override certain option there is a `config.vault.get_override_examples()` helper:

```
In [1]: config.vault.get_override_examples()
{
    "NESTED_SECRET": {
        "PATH__TO__NESTED": '{"NESTED_SECRET": {"nested": "example_value"}}'
    },
    "SECRET": {
        "PATH__TO": '{"SECRET": "example_value"}'
    },
    "WHOLE_SECRET": {
        "PATH__TO": "{}"
    },
}
```

By default, when the evaluation will happen on a Vault secret, the environment will be checked first. If you don't need this behavior, it could be turned off with `try_env_first=False` option to the chosen backend:

```
# app_name/settings/__init__.py
from konfetti import Konfig, AsyncVaultBackend

config = Konfig(vault_backend=AsyncVaultBackend("your/prefix", try_env_first=False))
```

Disabling access to secrets

If you want to forbid any access to Vault (e.g. in your tests) you can set `KONFETTI_DISABLE_SECRETS` environment variable with `1` / `on` / `true` / `yes`.

```
In [1]: import os
In [2]: from app_name.settings import config
In [3]: os.environ["KONFETTI_DISABLE_SECRETS"] = "1"
In [4]: (await config.get_secret("path/to"))["key"]
...
RuntimeError: Access to secrets is disabled. Unset KONFETTI_DISABLE_SECRETS variable␣
↪to enable it.
```

Caching

Vault values could be cached in memory:

```
config = Konfig(vault_backend=AsyncVaultBackend("your/prefix", cache_ttl=60))
```

By default, caching is disabled.

Retries

Vault calls would be retried in case of network issues, by default it is 3 attempts or up to 15 seconds.

This behavior could be changed via vault backend options

```
config = Konfig(vault_backend=AsyncVaultBackend("your/prefix", max_retries=3, max_
↪retry_time=15))
```

Also it is possible to pass retrying object with custom behavior, e.g. tenacity.Retrying or tenacity.AsyncRetrying:

1.3.3 Lazy options

If there is a need to calculate config options dynamically (e.g., if it depends on values of other options) `konfetti` provides `lazy`:

```
from konfetti import lazy

LAZY_LAMBDA = lazy(lambda config: config.KEY + "/" + config.SECRET + "/" + config.
↪REQUIRED)


@lazy("LAZY_PROPERTY")
def lazy_property(config):
    return config.KEY + "/" + config.SECRET + "/" + config.REQUIRED
```

1.4 Testing

It is usually a good idea to use a slightly different configuration for tests (disabled tracing, sentry, etc.).

```
export KONFETTI_SETTINGS=app_name.settings.tests
```

It is very useful to override some config options in tests. `Konfig.override` will override config options defined in the settings module. It works as a context manager or a decorator to provide explicit setup & clean up for overridden options.

```
from app_name.settings import config

# DEBUG will be `True` for `test_everything`
@config.override(DEBUG=True)
def test_everything():
    # DEBUG will be `False` again for this block
    with config.override(DEBUG=False):
        ...
```

Overrides could be nested, and deeper level has precedence over all levels above:

```
from app_name.settings import config

@config.override(FOO=1, BAR=2)
def test_many_things():
    with config.override(BAR=3):
        assert config.FOO == 1
        assert config.BAR == 3
    # As it was before
    assert config.BAR == 2
```

Also, override works for classes (including inherited from `unittest.TestCase`):

```
@config.override(INTEGER=123)
class TestOverride:

    def test_override(self):
        assert config.INTEGER == 123

    @config.override(INTEGER=456)
    def test_another_override(self):
        assert config.INTEGER == 456

def test_not_affected():
    assert config.INTEGER == 1
```

NOTE. `setup_class/setUp` and `teardown_class/tearDown` methods will work with `override`.

`konfetti` includes a `pytest` integration that gives you a fixture, that allows you to override given config without using a context manager/decorator approach and automatically rollbacks changes made:

```
import pytest
from app_name.settings import config
from konfetti.pytest_plugin import make_fixture

# create a fixture. the default name is "settings",
# but could be specified via `name` option
make_fixture(config)

@pytest.fixture
def global_settings(settings):
    settings.INTEGER = 456


@pytest.mark.usefixtures("global_settings")
def test_something(settings):
    assert settings.INTEGER == 456
    assert config.INTEGER == 456

    # fixture overriding
    settings.INTEGER = 123
    assert settings.INTEGER == 123
    assert config.INTEGER == 123

    # context manager should work as well
    with settings.override(INTEGER=7):
        assert settings.INTEGER == 7
```

(continues on next page)

(continued from previous page)

```
        assert config.INTEGER == 7

    # Context manager changes are rolled back
    assert settings.INTEGER == 123
    assert config.INTEGER == 123


# This test is not affected by the fixture
def test_disable(settings):
    assert config.INTEGER == 1
    assert settings.INTEGER == 1
```

NOTE. It is forbidden to create two fixtures from the same config instances.

1.5 Extending configuration

Sometimes configuration is distributed to multiple places - python modules, JSON files, etc. To handle everything seamlessly there is a couple of methods:

```
config = Konfig()
config.extend_with_object("path.to.module")
config.extend_with_object({"KEY": "value"})
config.extend_with_json("/path/to.json")
```

Importable strings and JSON files will be loaded lazily on the first access.

1.6 Extras

The environment variable name could be customized via `config_variable_name` option:

```
config = Konfig(config_variable_name="APP_CONFIG")
```

Alternatively, it is possible to specify class-based settings:

```
from konfetti import env, vault


class ProductionSettings:
    VAULT_ADDR = env("VAULT_ADDR")
    VAULT_TOKEN = env("VAULT_TOKEN")

    DEBUG = env("DEBUG", default=False)
    DATABASE_URI = vault("path/to/db")
```

It possible to load the whole config and get its content as a dict:

```
In [1]: await config.asdict()
{
    "ENV": "env value",
    "KEY": "static value",
    "SECRET": "secret_value",
}
```

If you need to validate that certain variables are present in the config, there is `require`:

```
In [1]: config.require("SECRET")
...
MissingError: Options ['SECRET'] are required
```

Or to check that they are defined:

```
In [1]: "SECRET" in config
True
```

1.7 Django integration

To magically convert `django.conf.settings` into konfetti config object, add this to the very end of your project settings module.

```
config = install(
    __name__,
    vault_backend=VaultBackend("your/prefix")
)
```

Having this will allow the application to use settings, defined via `vault`, `env` and other types from konfetti.

1.8 Flask extension

There is an extension for `Flask` that replaces `Flask.config` with `Konfig` instance and adds all `konfetti` features to your `app.config`.

```
from flask import Flask
from konfetti.contrib.flask import FlaskKonfig
from .settings import config

app = Flask(__name__)
FlaskKonfig(app, config, CUSTOM_OPTION=42)

...

# Taken from Vault
assert app.config.SECRET == "value"
# Manually specified
assert app.config.CUSTOM_OPTION == 42
```

CHAPTER 2

Configuration 101

There are a couple of principles that will help you to avoid problems when you specify or use your configuration.

2.1 Do not access configuration on the module level

Do this:

```
from app_name.settings import config

def get_redis_client():
    return StrictRedis.from_url(config.REDIS_URL)
```

Instead of this:

```
from redis import StrictRedis
from app_name.settings import config

cache_redis = StrictRedis.from_url(config.REDIS_URL)
```

In this case on each usage the redis client will be re-evaluated, which might be not good for performance reasons.

As an alternative you could have a global Redis instance by using `python-lazy-object-proxy`:

```
pip install lazy-object-proxy
```

```
import lazy_object_proxy

...

cache_redis = lazy_object_proxy.Proxy(get_redis_client)
```

NOTE. Do not forget to clean up shared resources when it is needed, usually on the application / testcase teardown.

2.1.1 Why?

Accessing configuration on the module level leads to side-effects on imports, this fact could produce unrelated errors when you run your test suite:

- Simple unit tests will fail due to lack of configuration options or Vault unavailability;
- Slow tests due to config initialization and long network calls (they could time out as well);

Having your config access lazy will prevent many for those cases because that code branches won't be executed on imports and will not affect your test suite.

CHAPTER 3

Contributing

3.1 Code formatting

In order to maintain code formatting consistency we use black to format the python files. A pre-commit hook that formats the code is provided but it needs to be installed on your local git repo, so...

In order to install the pre-commit framework run `pip install pre-commit` or if you prefer homebrew `brew install pre-commit`

Once you have installed pre-commit just run `pre-commit install` on your repo folder

If you want to exclude some files from Black (e.g. automatically generated database migrations or test snapshots) please follow instructions for pyproject.toml

3.2 Testing

To run all tests:

```
docker run -p 8200:8200 -d --cap-add=IPC_LOCK -e 'VAULT_DEV_ROOT_TOKEN_ID=test_root_
↪token' vault:0.9.6
tox -p all
```

Note that tox doesn't know when you change the `requirements.txt` and won't automatically install new dependencies for test runs. Run `pip install tox-battery` to install a plugin which fixes this silliness.

It also possible to run tests via docker-compose that will start up all required environment:

```
$ make docker-test
```

or alternatively:

```
$ docker-compose -f docker-compose-tests.yml run konfetti
```

CHAPTER 4

Changelog

4.1 Unreleased

4.2 ‘0.8.0‘_ - 2020-01-15

4.2.1 Added

- Add keys to environment variable name before search in Vault.

4.2.2 Fixed

- Change Flask KonfigProxy to behave like Flask config.

4.3 0.7.2 - 2019-07-18

4.3.1 Fixed

- Modification of vault variables in `[]` access. #51

4.4 0.7.1 - 2019-07-17

4.4.1 Fixed

- Evaluation of config options that are dictionaries. #47

4.5 0.7.0 - 2019-07-15

4.5.1 Added

- Retries for Vault backends. #12
- Alternative constructors for `Konfig`
- Add configuration extending. #34
- Add casting for container types. #35
- Add `userpass` authentication method for vault #18
- Django integration. #31
- Flask integration. #32

4.6 0.6.0 - 2019-06-30

- Initial public release

CHAPTER 5

Indices and tables

- genindex
- modindex
- search