App Enabler Documentation

Release 0.3.0

Nephila

Nov 09, 2023

CONTENTS

CHAPTER

ONE

APP ENABLER

1.1 Description

PoC autoconfigurator for django applications

`django-app-enabler` goal is to reduce the configuration of a django application to a one command operation to ease using django applications, both for newcomers and expert developers.

As configuring a django application can be both boring (as 90% are the usual steps editing `settings.py` and `urls.py`) and complex (as it's easy to overlook one vital configuration parameter), replacing this with a single command sounds like a real benefit.

1.1.1 Key points

- zero-knowledge tool to enable and configure django applications in a django project
- rely on specification file shipped by the target application to patch django project configuration
- not a replacement for existing package or dependencies managers (pip / poetry / pipenv / ...)

1.1.2 Caveats

- Project is currently just a proof of concept
- No formal specification or documentation exist (yet) for addon configuration file
- A lot of restrictions regarding the `settings.py` and `urls.py` files are currently in place
- Not all standard django settings options are currently supported

See usage for more details.

1.1.3 Compatible packages

Up-to-date list of compatible packages

CHAPTER

TWO

USAGE

`django-app-enabler` allow application supporting *Addon configuration specification* to be installed and configured automatically in the current django project.

2.1 Installation

```
pip install django-app-enabler
```

2.2 Commands

- *apply <path_to_json> <path_to_json>*: Apply configuration from json files
- *enable <module_name>*: Configure an application
- *install <package-name>*: Install and configure an application

2.3 Sample execution flow

```
django-enabler install djangocms-blog~=1.2.1
python manage.py migrate
```

After this the django application is configured and functional.

Additional configuration steps might be required according to the application features and support level and must be documented by the application itself.

Alternatively you can execute the module itself:

```
python -mapp_enabler install djangocms-blog~=1.2.1
```

2.4 Application configuration

The core of `django-app-enabler` is its Django configuration patching engine.

The general concept is that once a django package is installed, `app-enabler` can be run from the project root and the project is automatically updated with the minimal configuration required by the application to run (or any superset of this definition).

Applied configurations are declared by the target application in a *addon.json* file included in the python package.

Example:

```
django-enabler enable djangocms_blog
```

See *Limitations* for limitations and caveats.

2.5 Apply configurations

`django-app-enabler` can also apply configuration from arbitrary json files not included in any Django application.

Each configuration file must comply with *Extra configuration files specifications*.

Note: Django `settings` and `urlconf` are patched unconditionally. No attempt to verify that applications declared in `installed_apps` or added to the `urlconf` are available in the virtualenv is made.

Example:

```
django-enabler apply /path/to/config1.json /path/to/config2.json
```

See *Limitations* for limitations and caveats.

2.6 Application Installation

As a convenience `django-app-enabler` can execute `pip install` on your behalf, though step this is not required.

The `install` command will both install the package and enable it.

Installation is executed via the `install` command which a

```
django-enabler install djangocms-blog~=1.2.0
```

Note: `django-app-enabler` is not intended as a replacement (or sidekick) of existing package / dependencies manager. The installation step is only intended as a convenience command for those not sticking to any specific workflow. If you are using anything than manual `pip` to install packages, please stick to it and just use *Application configuration*.

CHAPTER

THREE

LIMITATIONS

Paching features have currently the following limitations:

3.1 settings.py

- Only single file `settings.py` are currently supported. In case you are using splitted settings, the only way to use `django-app-enabler` is to have at least an empty `INSTALLED_APPS` list in the settings file declared in `DJANGO_SETTINGS_MODULE`.
- Settings with literal or "simple" lists and dictionaries (like `CACHE`, `DATABASES`, `AUTH_PASSWORD_VALIDATORS`) are supported, the most notable exception is `TEMPLATES` in which you cannot add / replace options in a single template engine. Any custom setting is supported.
- While extra requirements will be installed when including them in the package argument (as in `djangocms-blog[search]`), they will not be added to `INSTALLED_APPS` and they must be added manually after command execution.

3.2 urls.py

- Only single file `urls.py` are currently supported. In case you are using splitted settings, the only way to use `django-app-enabler` is to have at least an empty `urlpatterns` list in the `settings.ROOT_URLCONF` file.

CHAPTER

FOUR

ADDON CONFIGURATION SPECIFICATION

`django-app-enabler` support can be enabled by adding a *addon.json* to any django application (see below for the structure).

See *Limitations* for limitations and caveats.

4.1 addon.json

`addon.json` is the only configuration file needed to support `django-app-enabler` and it **must** provide at least the minimal setup to make the application up an running on a clean django project.

> **Warning:** The file must be included in root of the first (alphabetically) module of your application package. See *Packaging* for details.

4.2 Extra configuration files specifications

Extra configuration files (applied via *Apply configurations*) must conform to the same specifications below with two exceptions:

- all attributes are optional (i.e.: they can be completely omitted)
- the json file can contain a single object like for the `addon.json` case, or a list of objects conforming to the specifications.

4.2.1 Attributes

The following attributes are currently supported:

- `package-name` [**required**]: package name as available on PyPi;
- `installed-apps` [**required**]: list of django applications to be appended in the project `INSTALLED_APPS` setting. Application must be already installed when the configuration is processed, thus they must declared as package dependencies (or dependencies of direct dependencies, even if this is a bit risky);
- `urls` [optional]: list of urlconfs to be added to the project `ROOT_URLCONF`. List can be empty if no url configuration is needed or it can be omitted.

  Each entry in the list must be in the `[<patten>,<include-dotted-path>]` format:

  - `<pattern>` must be a `Django path()` pattern string, it can be empty (to add the urlconf to the root)

  - <include-dotted-path> must be a valid input for Django include() function;
- settings [optional]: A dictionary of custom settings that will be added to project settings verbatim;
- message [optional]: A text message output after successful completion of the configuration;

Attribute format

installed-apps and settings values can have the following formats:

- literal (string, int, boolean): value is applied as is
- dict with the following structure:
  - value: Any (required), the setting value
  - position: int, if set and the target setting is a list, value is inserted at position
  - next: str, name of an existing item before which the value is going to be inserted
  - key: str, in case value is a dictionary, the dictionary key to be used to match existing settings value for duplicates and to match the next value

4.2.2 Merge strategy

settings items not existing in the target project settings are applied without further changes, so you can use whatever structure is needed.

settings which already exists in the project and installed-apps configuration are merged with the ones already existing according to this strategy:

- setting does not exist -> custom setting is added verbatim
- setting exists and its value is a literal -> target project setting is overridden
- setting exists and its value is a list -> custom setting is merged:
  - if the custom setting is a literal -> its value is appended to the setting list
  - if it's a dictionary (see format above) ->
    - if next is defined, a value matching the next value is searched in the project setting and the custom setting value is inserted before the next element or at the top of the list if the value is not found; in case value (and items in the project settings) are dictionaries (like for example AUTH_PASSWORD_VALIDATORS), a key attribute must be provided as a lookup key;
    - if position is defined, the custom setting value is inserted at that position;

In any case, if a value is already present, is not duplicated and is simply ignored.

4.2.3 Sample file

```
{
    "package-name": "djangocms-blog",
    "installed-apps": [
        "filer",
        "easy_thumbnails",
        "aldryn_apphooks_config",
        "parler",
```

(continues on next page)

(continued from previous page)

```
        "taggit",
        "taggit_autosuggest",
        "meta",
        "djangocms_blog",
        "sortedm2m"
    ],
    "settings": {
        "META_SITE_PROTOCOL": "https",
        "META_USE_SITES": true,
        "MIDDLEWARE": [
            "django.middleware.gzip.GZipMiddleware",
            {"value": "django.middleware.http.ConditionalGetMiddleware", "position": 2},
            {
                "value": "django.middleware.locale.LocaleMiddleware",
                "next": "django.middleware.common.CommonMiddleware",
            },
        ],
        "AUTH_PASSWORD_VALIDATORS": [
            {
                "value": {
                    "NAME": "django.contrib.auth.password_validation.
↪NumericPasswordValidator",
                },
                "next": "django.contrib.auth.password_validation.
↪UserAttributeSimilarityValidator",
                "key": "NAME",
            },
        ],
    },
    "urls": [
        ["", "djangocms_blog.taggit_urls"]
    ],
    "message": "Please check documentation to complete the setup"
}
```

4.3 Packaging

TBA

CHAPTER

FIVE

PLANNED FEATURES

- Support extra-requirements issue-6
- Support Django settings split in multiple files issue-7
- Support Django urlconf split in multiple files issue-8

CHAPTER

SIX

API

6.1 Commands

6.1.1 django-enabler

Click entrypoint.

```
django-enabler [OPTIONS] COMMAND [ARGS]...
```

Options

--verbose

apply

Apply configuration stored in one or more json files.

CONFIG_SET: Path to configuration files

param click.core.Context context
Click context

param list config_set
list of paths to addon configuration to load and apply

```
django-enabler apply [OPTIONS] [CONFIG_SET]...
```

Arguments

CONFIG_SET
Optional argument(s)

enable

Enable the application in the current django project.

APPLICATION: Application module name (example: 'djangocms_blog')

param click.core.Context context
Click context

param str application
python module name to enable. It must be the name of a Django application.

```
django-enabler enable [OPTIONS] APPLICATION
```

Arguments

APPLICATION
Required argument

install

Install the package in the current virtualenv and enable the corresponding application in the current project.

PACKAGE: Package name as available on PyPi, or rather its requirement string.
Accepts any PEP-508 compliant requirement.
Example: "djangocms-blog~=1.2.0"

param click.core.Context context
Click context

param str package
Name of the package to install

param str pip_options
Additional options passed to pip

```
django-enabler install [OPTIONS] PACKAGE
```

Options

--pip-options <pip_options>
Additional options passed as is to pip

Arguments

PACKAGE
Required argument

6.2 CLI

app_enabler.enable.**apply_configuration**(*application_config: Dict[str, Any]*)
Enable django application in the current project

Parameters
application_config (*dict*) – addon configuration

app_enabler.enable.**apply_configuration_set**(*config_set: List[Path]*, *verbose: bool = False*)
Apply settings from the list of input files.

Parameters

- **config_set** (*list*) – list of paths to addon configuration to load and apply
- **verbose** (*bool*) – Verbose output (currently unused)

app_enabler.enable.**enable_application**(*application: str*, *verbose: bool = False*)
Enable django application in the current project

Parameters

- **application** (*str*) – python module name to enable. It must be the name of a Django application.
- **verbose** (*bool*) – Verbose output (currently unused)

app_enabler.enable.**output_message**(*message: str*)
Print the given message to stdout.

Parameters
message (*str*) – Success message to display

app_enabler.enable.**verify_installation**(*settings: LazySettings*, *application_config: Dict[str, Any]*) → bool
Verify that package installation has been successful.

Parameters

- **settings** (*django.conf.LazySettings*) – Path to settings file
- **application_config** (*dict*) – addon configuration

app_enabler.install.**get_application_from_package**(*package: str*) → str | None
Detect the first in alphabetical order module provided by a package.

This approach is a bit simplistic, but as we only need this to get the `addon.json` file specified by this package, we can easily enforce this restriction.

Parameters
package (*str*) – package name (or rather its requirement string). It can be anything complying with PEP508

Returns
main (first) module name; if `None`, package is not available in the current virtualenv

app_enabler.install.**install**(*package: str*, *verbose: bool = False*, *pip_options: str = ''*)

Install the package.

Installation is done via pip executed as a subprocess to ensure maximum compatibility.

Parameters

- **package** (*str*) – Package name
- **verbose** (*bool*) – Verbose output
- **pip_options** (*str*) – Additional options passed to pip

6.3 Loaders

app_enabler.django.**get_settings_path**(*setting: LazySettings*) → str

Get the path of the django settings file path from the django settings object.

Parameters
setting (*django.conf.LazySettings*) – Django settings object

Returns
path to the settings file

app_enabler.django.**get_urlconf_path**(*setting: LazySettings*) → str

Get the path of the django urlconf file path from the django settings object.

Parameters
setting (*django.conf.LazySettings*) – Django settings object

Returns
path to the settings file

app_enabler.django.**load_addon**(*module_name: str*) → Dict[str, Any] | None

Load addon configuration from json file stored in package resources.

If addon has no configuration, return `None`.

Parameters
module_name (*str*) – name of the python module to load as application

Returns
addon configuration

6.4 Patchers

class app_enabler.patcher.**DisableExecute**

Patch the `manage.py` module to remove the execute_from_command_line execution.

visit_Expr(*node: AST*) → Any

Visit the `Expr` node and remove it if it matches `'execute_from_command_line'`.

app_enabler.patcher.**monkeypatch_manage**(*manage_file: str*) → code

Patch `manage.py` to be executable without actually running any command.

By using ast we remove the `execute_from_command_line` call and add an unconditional call to the main function.

Parameters
manage_file (*str*) – path to manage.py file

Returns
patched manage.py code

app_enabler.patcher.**setup_django**()

Initialize the django environment by leveraging `manage.py`.

This works by using `manage.py` to set the `DJANGO_SETTINGS_MODULE` environment variable for `django.setup()` to work as it's unknown at runtime.

This should be safer than reading the `manage.py` looking for the written variable as it rely on Django runtime behavior.

Manage.py is monkeypatched in memory to remove the call "execute_from_command_line" and executed from memory.

app_enabler.patcher.**update_setting**(*project_setting: str*, *config: Dict[str, Any]*)

Patch the settings module to include addon settings.

Original file is overwritten. As file is patched using AST, original comments and file structure is lost.

Parameters

- **project_setting** (*str*) – project settings file path
- **config** (*dict*) – addon setting parameters

app_enabler.patcher.**update_urlconf**(*project_urls: str*, *config: Dict[str, Any]*)

Patch the `ROOT_URLCONF` module to include addon url patterns.

Original file is overwritten. As file is patched using AST, original comments and file structure is lost.

Parameters

- **project_urls** (*str*) – project urls.py file path
- **config** (*dict*) – addon urlconf configuration

CHAPTER

SEVEN

HISTORY

7.1 0.3.0 (2023-11-09)

7.1.1 Features

- Improve merge strategy to support all the basic standard Django settings (#5)
- Add support for external configuration json (#9)
- Upgrade to Django 3.2/4.2 (#32)
- Switch to Coveralls Github action (#56)
- Migrate to bump-my-version (#58)

7.2 0.2.0 (2020-12-27)

7.2.1 Features

- Add CLI utility (#20)

7.2.2 Bugfixes

- Close resource_stream file pointer (#19)
- Fix importing include multiple times in urlconf (#21)
- Add test to verify no multiple urlconf are added (#25)

7.3 0.1.1 (2020-12-21)

7.3.1 Features

- Add codeql action (#15)

7.3.2 Bugfixes

- Fix errors with urlconf patching (#17)

7.4 0.1.0 (2020-12-20)

Initial release

7.4.1 Features

- Add install command (#1)
- Add tests (#2)
- Add support for message addon config parameter (#11)

7.4.2 Improved Documentation

- Improve documentation (#1)

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

C

D

E

G

I

L

M

O

P

S

U

V