cookiecutter-bireli

Release 0.3.6

David Thenon

Jun 05, 2023

CONTENTS

Bireli is a Django project template with Cookiecutter to produce a ready to start project.

It emphases on quality, modularity and modern stable stack.

Hint: This documentation is an ongoin work, not everything have been covered yet and it will continue to evolve.

CHAPTER ONE

FEATURES

- Development in a Python virtual environment with virtualenv and pip;
- Project include a `pyproject.toml` to store (almost) all backend tools configurations;
- Promote Test Driven Development with Pytest;
- Latest stable stack support;
- Frontend assets built with Node.js and managed with Webpack;
- Default shipped layout with Bootstrap5;
- Backend application architecture is modular through Project composer;
- Settings are managed with django-configurations;
- Internationalization and localization enabled;
- Include a set of main applications (CMS, blog, form builder, etc..) pre-configured;
- A Makefile with every useful commands.

CHAPTER TWO

DEPENDENCIES

All involved dependencies

- **Bireli:** `0.3.6`
- **Bireli-newapp:** `0.1.1`
- **Python:** `>=3.10`
- **Django:** `>=4.0,<4.1`
- **Project-composer:** `>=0.7.0,<0.8.0`
- **Django-configurations:** `>=2.3.2`
- **Node:** `>=18.0.0`
- **Npm:** `>=8.0.0`
- **Bootstrap:** `5.2.0`
- **Webpack:** `^5.50.0`

Note: `Bireli` and `Bireli-newapp` are not involved in a project once it has been created.

CHAPTER

THREE

LINKS

- Read the documentation on Read the docs;
- Clone it on its Github repository;

CHAPTER

FOUR

SUMMARY

4.1 Create a new project

Warning: You don't need this just to use a Bireli project, this is only for developers that need to create some fresh new projects.

To create a new project you just need to install Cookiecutter version >=2.1.0.

You may then use it from its repository URL:

```
cookiecutter https://github.com/sveetch/cookiecutter-bireli.git
```

Note: To speed up project creation you may install this cookie on your system, read *Install for development* to know how.

4.1.1 Options

Once invoked, cookiecutter will prompt your for some informations about your project.

You may pre define some options in your cookiecutter user configuration to avoid to input them each time you use this cookie.

4.1.2 Result

Cookiecutter will create a new directory named after your project name. You can enter into its directory and install it locally with `make install` (see *Project install* for details).

4.2 Project install

Note: This document is about default procedure for a freshly created project. Some projects may have been changed by developers to involve less or more requirements, tasks and configurations.

Commonly a project should documentate everything for their specific needs but it is out of scope of Bireli documentation.

4.2.1 System requirements

A project will requires *Python*, pip, virtualenv, *GNU make* and a recent *Node.js* already installed and some system packages for installing and running.

Lists below are the required basic development system packages and some other optional ones.

Basic requirements

Warning: Package names may differ depending your system.

- Git;
- Python `>=3.10`;
- `python-dev`;
- `python-virtualenv`;
- `gettext`;
- `gcc`;
- `make`;
- `libjpeg`;
- `libcairo2`;
- `zlib`;
- `libfreetype`;

Hint: If your system does not have the right Python version as the default one, you should use something like pyenv to install it and then use `pyenv local` to set the correct project Python version to use.

On Linux distribution

You will install them from your common package manager like `apt` for Debian based distributions:

```
apt install python-dev python-virtualenv gettext gcc make libjpeg libcairo2 zlib␣
↪libfreetype
```

On macOS

Recommended way is to use `brew` utility for system packages, some names can vary.

On Windows

Not supported, you probably can install some needed stuff but with some works on your own.

Optional requirements

These ones are common extra requirements that some projects may use. You don't need to take care of them for now.

For Postgresql client driver (psycopg2)

- `libpq`;

For Mysql client driver

- `libmysqlclient-dev`;

For M2Crypto

- `swig`;

For Graphviz

- `graphviz`;
- `libgraphviz-dev`;
- `graphviz-dev`;

4.2.2 Local deployment

A created project can be installed using a simple Makefile task:

```
make install
```

Now you need to build the frontend assets:

```
make frontend
```

When finished your project is ready to run.

4.2.3 Initial data

A new installed project is empty from any content, however a task exists to create some initial data for main components:

```
make initial-data
```

This will creates a user with username `admin` and password `ok`.

If you don't want any initial data, you will need at least a super user to reach the admin:

```
make superuser
```

4.2.4 Quickstart

Once you already installed a Bireli project, you should have all needed requirements and you may just quickly do everything in a single command:

```
make install frontend initial-data
```

Or:

```
make install frontend superuser
```

4.2.5 Upgrades

Later if a project introduces a new package or newer packages versions, you may use the following commands to upgrade your local install.

To upgrade backend install:

```
make install-backend
```

To upgrade frontend install:

```
make install-frontend
```

Warning: Don't use the task `install` to upgrade your install, it has been made for a fresh new install and include some other tasks that are longer to run and that could also lost some of your changes.

4.2.6 Cleaning

If you need to reset your local install you may use the following command:

```
make clean
```

However this will remove everything even your local data. If you just need to clean some parts of your install, see Makefile help for all the specific cleaning tasks.

4.2.7 Production deployment

This is out of scope of Bireli because there is just too many ways to deploy a project, you will have to add this layer on yourself into your project.

4.3 Makefile

A project contains a Makefile to achieve all the common tasks, use its help to know about every available task:

```
make help
```

4.3.1 Tasks

The following list is a summary of important tasks, use them like `make TASKNAME`.

requirements
: To build base requirements file for enabled applications from composition manifest.

  This is only to use when you change requirements files from repository application or when you change enabled application from composer manifest.

install
: To perform a new install with both backend and frontend.

install-backend
: To install or upgrade backend requirements with Virtualenv and Pip.

install-frontend
: To install or upgrade frontend requirements with Npm.

freeze-dependencies
: To write a frozen.txt file with installed dependencies versions

clean
: To clean EVERYTHING (WARNING: you cannot recovery from this).

  This use all the available clean tasks, see Makefile help to know about them.

check
: To run all following check tasks in an accurate order to ensure debugging level.

check-composer
: To run Composer checking on its configuration and display an helpful report.

check-django
: To run Django System check framework. This is the most simple way to check about your project health but it won't go deeper like tests can do.

check-migrations
: To check for pending application migrations. It does not write anything, just output all pending migration Django found from your project.

  This is useful when you are working on models since every tiny change can require a migration.

run
: To run Django development server on your local network interface on port `8001`.

  By default you will be able to reach it with `http://localhost:8001/`.

migrate
: To apply pending models migrations. This is to run when you have created new migrations or when you updated your local install which can bring some model changes.

superuser
: To quickly create a new superuser for Django admin from commandline. Obviously once you already have a superuser you may use the Django admin to create new users.

initial-data
To load initial data for enabled applications. You should not run it twice on the same database.

new-app
To create a new project application properly structured and configured using template bireli-newapp;

css
To build CSS for development environnement, this means without any optimization.

watch-css
To launch watcher CSS for development environnement. On every Sass sources change a build will be performed to update CSS.

js
To build distributed Javascript for development environnement.

watch-js
To launch watcher for Javascript sources for development environnement.

frontend
To build frontend assets from sources (CSS and JS) for development environnement.

po
To update every PO files from composition apps, django apps and project code and templates for enabled languages.

This won't create the locale directory for new enabled languages from settings, you must boot it yourself.

Saying to add French language, first you need to add `("fr", "French"),` to `settings.LANGUAGES`. Then after you will run a command like this:

```
.venv/bin/python manage.py makemessages --keep-pot --no-obsolete --locale fr
```

Never copy another language directory and rename it to your new locale name, it will miss some specific locale options added by gettext (like plural formula).

mo
To build MO files from existing project PO files.

flake
To launch Flake8 checking on project backend code.

test
To launch project test suite using Pytest.

quality
To launch all quality tasks, any failure will stop its execution.

4.4 Architecture

4.4.1 Composition

Bireli strongly stands on Project composer to structure its main parts (settings, urls and requirements). You will need to properly understand Project composer before to properly work on a project.

The *Workflow* document from Project composer documentation contains a diagram exemple of resumed workflow for a Django project.

Details

The **composer configuration lives in the** `pyproject.toml` file in sections named `tool.project_composer[.**]`. Commonly you will only have to care about the option `collection` where is enabled all compose applications.

Note: Sections `tool.project_composer[.**]` assemble many options which assemble the composer configuration and that is called the *Manifest*.

The **collection is a list of module directory names from** `composition_repository/`.

You rarely have to edit the environment settings from `project/settings` because their purpose is only to override base settings for very specific environment needs.

All the **Django builtins settings are located in the compose application** `django_builtins`. And in the same idea, **each project application settings will be in their compose application**.

4.4.2 Structure

Here below we will explain the default project structure, there is many more files and directories but for a better explanation we will only focus on important parts.

```
.
├── composition_repository/
│   ├── django_builtins/
│   └── sample_app/
├── django-apps/
│   ├── project_utils/
│   └── sample_app/
├── frontend/
│   ├── js/
│   ├── scss/
│   ├── package.json
│   └── webpack.config.js
├── Makefile
├── project/
│   ├── settings/
│   ├── static-sources/
│   ├── templates/
│   └── urls.py
├── pyproject.toml
├── requirements/
└── tests/
```

composition_repository/
This is the directory which holds the applications configurations that will compose the project. These applications are enabled or not from the `collection` list from `pyproject.toml`.

django-apps/
This is the directory which hold the applications code (models, view urls, views, etc..).

frontend/
Everything related to frontend assets is defined and built from there.

- Javascript sources are in `js/`;

- Sass sources are in `scss/`;
- Frontend requirements are defined in `package.json`;
- Asset management is configured in `webpack.config.js`;

project/
This holds the Django project configuration and built assets.

- `settings/` store all the environment settings;
- `static-sources` will contains all built static to serve. It is not to mistake with `static` that is virtual directory that is only used in production so don't put anything there.
- `templates/` store all the project and applications templates;
- `urls.py` mount all the applications urls modules;

requirements/
This holds all *Environment Requirements*.

tests/
This is where to write all backend tests including project tests and all applications tests. No test in the applications directories is allowed because we want to store them in the same place.

pyproject.toml
The project backend manifest contains the Project composer manifest, versionning and many development tool configurations.

4.5 Backend

4.5.1 Backend base dependencies

- **Python:** `>=3.10`
- **Django:** `>=4.0,<4.1`
- **Project-composer:** `>=0.7.0,<0.8.0`
- **Django-configurations:** `>=2.3.2`

4.5.2 Database

A project is meant to work with different database drivers, at least PostgreSQL and SQlite. SQlite is used for development and test environments. PostgreSQL is used in all other deployment, especially production.

4.5.3 Settings

Project settings are defined using the django-configurations way, it means within a class. There is no more monolithic settings files.

There is two settings files kinds:

Application settings
Each application can have a settings file located in application module in composer repository. This is where you will configure all application settings.

Environment settings
They are located in `project/settings/` and their goal is to override some application settings to fit some special environment requirements.

Local settings

A special environment settings can be used to add or override settings for your own local purpose only. This is useful when you need to use some special things like debugging tools, database configuration, etc..

This settings file does not exists yet and you must create it to `project/settings/localsettings.py`.

Note: Alike all project settings files (from composer applications and environments), this local settings file has to be done for the django-configurations way.

Warning: This settings file must never be committed to the project repository since it is for your own local usage.

Basic

This example is only for basic apps which only need some settings to work.

Here we just enable django-extensions and disable cache. Its content should be something like:

```python
from .development import Development


class LocalEnv(Development):
    # Disable every cache in local development
    CACHES = {
        "default": {
            "BACKEND": "django.core.cache.backends.dummy.DummyCache",
        }
    }

    @classmethod
    def post_setup(cls):
        super(LocalEnv, cls).post_setup()

        cls.INSTALLED_APPS.extend([
            "django_extensions",
        ])
```

There can only be a single class and it must be named `LocalEnv` and inherits from `Development` class.

Advanced

Sometime an application needs some settings and to add some urls. Let's demonstrate it with configuration for both django-extensions and django-debug-toolbar.

First the settings file:

```python
from .development import Development


class LocalEnv(Development):
    ROOT_URLCONF = "project.localurls"

    INTERNAL_IPS = [
        "localhost",
    ]

    DEBUG_TOOLBAR_PANELS = [
        #"debug_toolbar.panels.history.HistoryPanel",
        "debug_toolbar.panels.versions.VersionsPanel",
        "debug_toolbar.panels.timer.TimerPanel",
        "debug_toolbar.panels.settings.SettingsPanel",
        "debug_toolbar.panels.headers.HeadersPanel",
        "debug_toolbar.panels.request.RequestPanel",
        "debug_toolbar.panels.sql.SQLPanel",
        "debug_toolbar.panels.staticfiles.StaticFilesPanel",
        "debug_toolbar.panels.templates.TemplatesPanel",
        "debug_toolbar.panels.cache.CachePanel",
        #"debug_toolbar.panels.signals.SignalsPanel",
        #"debug_toolbar.panels.redirects.RedirectsPanel",
        #"debug_toolbar.panels.profiling.ProfilingPanel",
    ]

    # Disable every cache in local development
    CACHES = {
        "default": {
            "BACKEND": "django.core.cache.backends.dummy.DummyCache",
        }
    }

    @classmethod
    def setup(cls):
        super(LocalEnv, cls).setup()

        cls.MIDDLEWARE = [
            "debug_toolbar.middleware.DebugToolbarMiddleware",
        ] + cls.MIDDLEWARE

    @classmethod
    def post_setup(cls):
        super(LocalEnv, cls).post_setup()

        cls.INSTALLED_APPS.extend([
```

(continues on next page)

(continued from previous page)

```
            "django_extensions",
            "debug_toolbar",
        ])
```

As you can see we define a new main `urls.py` file that will inherit from the base main one and add some custom urls. Let's create it to `project/localurls.py`:

```
from django.urls import include, path

from project.urls import urlpatterns


urlpatterns = [
    path('__debug__/', include('debug_toolbar.urls')),
] + urlpatterns
```

Alike the local settings file, this file must never be commited to the repository.

4.5.4 Developing a new application

A Makefile task exists to help you to quickly start a new application into your project, just use:

```
make new-app
```

It will prompt you for a full title that will be used to build proper Python names (using slugify) and generate everything (composer application module, Django application module, etc..).

Once done the command outputs a resume and a some help to enable your new application.

4.5.5 Add a new third party application

To add a new package for an already enabled application just put it in application requirement file and configure it in its settings file. For example, a CMS plugin should live in the CMS application settings.

But sometime a third party application may be shared by many applications, in this case it will needs its own composer application module.

You may copy an other application module and edit it or use the command from *Developing a new application* and just keep the composer application folder.

4.5.6 Environment Requirements

Environment requirements are divided into multiple files because each environment may not use everything and so does not install everything.

Warning: Don't edit these files and prefer to add your requirements through a composer application to keep project well structured.

`composer.txt`

This is for the composer requirement itself which is appart from the backend base requirements.

It is required by every environment.

`base_template.txt`
: This is a template used by composer to generate again the base requirements file, do not edit it.

  It is not required directly by any environment.

`base.txt`
: This is the base project requirements. Don't write anything in it since it generated from composer, all you changes will be lost definitively.

  It is required by every environment.

`development.txt`
: This is for requirements used to run test and other quality check.

  It is required by environments that need to run tests and quality check.

`production.txt`
: This is for requirements used to serve project, specify a proper SGBD driver, etc..

  It is only required by all "non-local" environments that need to serve and run project.

`codestyle.txt`
: This is extra requirements in local environment to check and apply linters on code.

  It is not required by any environment. However it is installed in local environment.

`toolbox.txt`
: This is extra requirements in local environment for some common helpful tools for debugging.

  It is not required by any environment. However it is installed in local environment.

Note: Project does not include configuration needed by extra requirements, especially the Django ones. You will need to enable and configure them through your *Local settings*.

4.6 Frontend

4.6.1 Frontend base dependencies

- **Node:** `>=18.0.0`
- **Npm:** `>=8.0.0`
- **Bootstrap:** `5.2.0`
- **Webpack:** `^5.50.0`

Asset management

Frontend assets are managed with Webpack and Django is aware of them through django-webpack-loader so you can load them from templates.

Compiled CSS from Sass sources are not managed from Webpack since there is currently no Sass compiler that are properly usable. So these CSS files are just loaded as simple static files.

Webdesign integration

Layout stylesheets (CSS) are built from Sass sources.

It is not allowed to use inline styles in templates and no *scoped* style from Javascript interfaces. The only source of truth for layout stylesheets are the Sass sources.

The build from Sass to CSS is performed from the frontend stack with node-sass. We still use node-sass because it's still the fastest compiler in Javascript.

Default project frontend use Bootstrap framework and all templates are made with its components.

Javascript interface

Default Javascript sources shipped in a project are basic and just load the Bootstrap components. Code sources are to be done for ES6 and jQuery is still available.

Logo and favicon

A project is generated with a default logo and favicon that you should change to fit to your project brand design.

Note than favicon is configured using a site manifest to cover multiple devices behaviors, you may build a new full site manifest from online tool like Favicon Generator (recommended).

4.7 Development

4.7.1 Install for development

First ensure you have pip, virtualenv packages installed and *GNU make* available on your system. Then type:

```
git clone https://github.com/sveetch/cookiecutter-bireli.git
cd cookiecutter-bireli
make install
```

Warning: You will need to keep your install up to date yourself opposed to the direct repository usage which always try to use the latest version.

Once installed you can create shortcut with a bash alias in your `.bash_aliases`:

```
alias cookdjango='/home/your/install/cookiecutter-bireli/.venv/bin/cookiecutter /home/
↪your/install/cookiecutter-bireli'
```

So you will just have to execute following command to create a new project:

```
cookdjango
```

Contribution

Every feature proposal and bug fixes must pass through a Pull request.

Note: To avoid managing main components versions through multiple files and miss some inconsistencies, main component versions are stored through private variables in cookiecutter template configuration file `cookiecutter.json`.

These variables are strings that must be valid requirement versions for Python package, except for the frontend components that must be valid versions for NPM.

4.8 History

4.8.1 Version 0.3.6 - 2023/05/22

- Upgraded `cmsplugin-blocks` to `==1.1.0` (fix critical bug that lost media during page publication);
- Added 404 and 500 templates;
- Fixed test settings to use `setup()` method instead of property to override `MEDIA_ROOT`;
- Cleaned `site_manifest.html` template;
- Fixed `freeze` Makefile task to export to `requirements/frozen.txt` instead of `requirements/requirements_freeze.txt`;
- Added *Basic requirements* new line about `libcairo2` in install documentation since it is a new requirement involved from library chain *django-filer* < *easy-thumbnail* < *reportlab*;
- Versionned main stylesheet using project version encoded in base64 for URL safety, it will be enough to prevent cache on production. However in development it won't really change anything since project version does not change often;
- Restored a proper CKEditor configuration with missing plugins CodeMirror, Youtube and Vimeo. Actually these plugins will be duplicated for `django-ckeditor` and `djangocms-text-ckeditor` because cookiecutter does not support symbolic link yet but a post hook will be done to resolve this;

4.8.2 Version 0.3.5 - 2023/04/28

- Added new applications in composer repository:
  - Added Lotus;
  - Added Cmsplugin-blocks;
  - Added Taggit;
  - Added DAL;
- Added a CMS toolbar for a shortcut link to Lotus articles, categories, Fobi, Taggit tags and Snippets;
- Added tasks for Black, Stylelint and djLint;
- Fixed issues from Stylelint on Sass sources;
- Fixed issues from djLint on templates;

4.8.3 Version 0.3.4 - 2023/03/28

- **Upgraded to Python>=3.10**;
- Removed usage of deprecated *setuptools private API* from `project/__init__.py` to get the project version. Instead it uses `tomli` to parse the project TOML file;
- Added `migrations` task to create all pending migrations from project applications;
- Added a common `pagination.html` template;
- Continued to improve documentation;
- Fixed `urls.py` from composer application which loaded url in the wrong order;
- Improved context process `site_metas` to include the project release version and included the version in skeleton into meta tag `generator`;
- Disabled fobi form template with Bootstrap5 to turn back to the simple theme since we cannot implement the Bootstrap5 form errors with fobi;
- Override `startapp` command with a new one which use bireli-newapp;
- Added more useful dev requirements files:
  - `codestyle` to apply and maintain codestyle quality;
  - `toolbox` for some debugging;
- Added Bireli logo as default project logo and favicon;
- Continued to improve documentation;

4.8.4 Version 0.3.3 - 2023/02/06

- Changed `check-migrations` task so it does not scan anymore for packaged app migrations, only the project ones from `django-apps`. This is to overcome issues CMS plugin apps that don't have yet a proper Django>=4.0 support, see issue #21 for details;
- Test environment settings no longer inherit from Development, instead some of Development settings have been copied to the Test settings;
- Fixed Composer check command which wrongly used resolver in lazy mode (leading to wrong order in output);
- Added feature for the optional local environment settings file `localsettings.py`;
- Moved `DOTENV` setting to `DjangoPaths` and make it conditional (to avoid confusing exception about Django apps and models) to Dotenv file existence;
- Fixed application settings and their `.env` sample. Now every setting that can be overwritten from Dotenv will use the default prefix `DJANGO_` such as a setting `FOO` is expected to be named `DJANGO_FOO` in Dotenv file;
- Fixed every applications settings files to explictely define `super()` arguments since it use `cls` and not `self` in setup methods;

4.8.5 Version 0.3.2 - 2023/01/30

- Started this history changelog;
- Started documentation;
- Added missing project directory `project/locale` and filled it with `en` and `fr` locale directories;
- Added missing locale directories `en` and `fr` with their PO;
- Fixed settings to remove translation for language names, they must always stand in their own language;