Colibris Framework

Nov 05, 2019

Contents

CHAPTER 1

Starting Your Project

The following variables are assumed:

- `VENVS` - the folder where you keep your python virtual environments (e.g. `~/.local/share/virtualenvs`)
- `PROJECT_NAME` - the name of your project (e.g. `my-project`)
- `PROJECTS_DIR` - the folder where you keep your projects (e.g. `~/Projects`)
- `PACKAGE` - the name of your main project's package
- `VERSION` - the version of your project

Create a virtual environment for your new project:

```
virtualenv ${VENVS}/${PROJECT_NAME} && source ${VENVS}/${PROJECT_NAME}/bin/activate
```

Install `colibris`:

```
pip install colibris
```

Go to your projects folder:

```
cd ${PROJECTS_DIR}
```

Prepare the project:

```
colibris-start-project ${PROJECT_NAME}
```

You can use a different skeleton template repository for your project:

```
colibris-start-project ${PROJECT_NAME} --skeleton git@github.com:myorganization/
↪microservice-skeleton.git
```

Your project folder will contain a package derived from your project name as well as various other stuff. You'll find a `manage.py` module in the project package, which is in fact the main script of your project.

You'll also find a `settings.py` module that you'll want to edit to adapt it to your project's needs.

The commands in this document assume you're in your project folder and you have your virtual environment correctly sourced, unless otherwise specified.

CHAPTER 2

Database

Choose a backend for the database, by setting the DATABASE variable in ${PACKAGE}/settings.py. By default, no database is enabled and the persistence layer is disabled.

2.1 SQLite Backend

In ${PACKAGE}/settings.py, set:

```
DATABASE = {
    'backend': 'colibris.persist.SQLiteBackend',
    'name': '/path/to/yourproject.db'
}
```

2.2 MySQL Backend

Make sure to have the mysqldb or pymysql python package installed.

In ${PACKAGE}/settings.py, set:

```
DATABASE = {
    'backend': 'colibris.persist.MySQLBackend',
    'name': 'yourproject',
    'host': '127.0.0.1',
    'port': 3316,
    'username': 'username',
    'password': 'password'
}
```

2.3 PostgreSQL Backend

Make sure to have the `psycopg2-binary` python package installed.

In `${PACKAGE}/settings.py`, set:

```
DATABASE = {
    'backend': 'colibris.persist.PostgreSQLBackend',
    'name': 'yourproject',
    'host': '127.0.0.1',
    'port': 5432,
    'username': 'username',
    'password': 'password'
}
```

CHAPTER 3

Models

Add your models by editing the `models.py` file:

```
nano ${PACKAGE}/models.py
```

CHAPTER 4

Schemas

Add your schemas by editing the `schemas.py` file:

```
nano ${PACKAGE}/schemas.py
```

CHAPTER 5

Views

Add your views by editing the `views.py` file:

```
nano ${PACKAGE}/views.py
```

5.1 APIView

For a simple API view, `colibris.views.APIView` can be used. Here is an example:

```python
class ItemsView(APIView):
    body_schema_class = ItemSchema
    query_schema_class = QuerySchema

    async def get(self):
        args = await self.get_validated_query()

        return web.json_response(args)

    async def post(self):
        data = await self.get_validated_body()

        return web.json_response(data)
```

Where `ItemSchema` and `QuerySchema` are simple marshmallow schemas.

5.2 ModelView

For a model based view, there is `colibris.views.ModelView` which has to be used together with at least one of: `ListMixin`, `CreateMixin`, `RetrieveMixin`, `UpdateMixin`, `DestroyMixin`. Here is an example of a model view which supports `GET` and `POST` methods:

```
class ItemsView(ModelView, ListMixin, CreateMixin):
    model = Model
    body_schema_class = ItemSchema
    query_schema_class = QuerySchema
```

For a basic RESTful resource there are predefined base views that can be used like this:

```
class ItemsView(ListCreateModelView):
    model = Model
    body_schema_class = ItemSchema

class ItemsDetailView(RetrieveUpdateDestroyModelView):
    model = Model
    body_schema_class = ItemSchema
```

5.2.1 Filtering

Filtering is also supported. A filter class will be created like this:

```
class ItemsFilter(ModelFilter):
    name = fields.String(field='name', operation=operators.EQ)

    class Meta:
        model = Item
        fields = {
            'name': (operators.EQ, operators.REGEXP, operators.NOT, operators.ILIKE),
            'count': (operators.GT, operators.GE, operators.LT, operators.LE)
        }


class ItemsView(ListCreateModelView):
    model = Model
    body_schema_class = ItemSchema
    filter_class = ItemsFilter
```

CHAPTER 6

Routes

Associate URL paths to views by editing the `routes.py` file:

```
nano ${PACKAGE}/routes.py
```

If you need routes for static files (not recommended for production), add your static prefix/path associations to `STATIC_ROUTES`.

CHAPTER 7

Authentication

Choose a backend for the authentication by setting the AUTHENTICATION variable in ${PACKAGE}/settings.py. By default, a null backend is used which associates each request with a dummy account.

7.1 JWT Backend

Make sure to have the pyjwt python package installed.

In ${PACKAGE}/settings.py, set:

```
AUTHENTICATION = {
    'backend': 'colibris.authentication.jwt.JWTBackend',
    'model': 'yourproject.models.User',
    'identity_claim': 'sub',
    'identity_field': 'username',
    'secret_field': 'password',
    'cookie_name': 'auth_token',
    'cookie_domain': 'example.com',
    'validity_seconds': 3600 * 24 * 30
}
```

The cookie_name property is optional and tells the backend to look for the token in cookies as well, in addition to the Authorization header.

The cookie_domain property is optional and configures the cookie domain.

The validity_seconds property is optional and configures the given validity for the token.

7.2 API Key Backend

In ${PACKAGE}/settings.py, set:

```
AUTHENTICATION = {
    'backend': 'colibris.authentication.apikey.APIKeyBackend',
    'model': 'yourproject.models.User',
    'key_field': 'secret',
}
```

CHAPTER 8

Authorization

Choose a backend for the authorization by setting the `AUTHORIZATION` variable in `${PACKAGE}/settings.py`. By default, a null backend is used, allowing everybody to perform any request.

8.1 Role Backend

In `${PACKAGE}/settings.py`, set:

```
AUTHORIZATION = {
    'backend': 'colibris.authorization.role.RoleBackend',
    'role_field': 'role'
}
```

8.2 Rights Backend

In `${PACKAGE}/settings.py`, set:

```
AUTHORIZATION = {
    'backend': 'colibris.authorization.rights.RightsBackend',
    'model': 'yourproject.models.Right',
    'account_field': 'user',
    'resource_field': 'resource',
    'operation_field': 'operation'
}
```

CHAPTER 9

Migrations

9.1 Create Migrations

To create migrations for your model changes, use:

```
./${PACKAGE}/manage.py makemigrations
```

You can optionally specify a name for your migrations:

```
./${PACKAGE}/manage.py makemigrations somename
```

9.2 Apply Migrations

To apply migrations on the currently configured database, use:

```
./${PACKAGE}/manage.py migrate
```

CHAPTER 10

HTTP

Start the HTTP server by running:

```
./${PACKAGE}/manage.py runserver
```

Then you can test it by pointing your browser to:

```
http://localhost:8888
```

CHAPTER 11

Initialization

You can add project-specific initialization code in the `init` function exposed by `app.py`:

```
nano ${PACKAGE}/app.py
```

CHAPTER 12

Cache

The caching mechanism is configured via the CACHE variable in ${PACKAGE}/settings.py. Caching is disabled by default.

12.1 Usage

To use the caching mechanism, just import it wherever you need it:

```
from colibris import cache
```

To set a value, use set:

```
cache.set('my_key', my_value, lifetime=300)
```

Later, you can get your value back:

```
my_value = cache.get('my_key', default='some_default')
```

You can invalidate a key using delete:

```
cache.delete('my_key')
```

12.2 Redis Backend

Make sure to have the redis python package installed.

In ${PACKAGE}/settings.py, set:

```
CACHE = {
    'backend': 'colibris.cache.redis.RedisBackend',
    'host': '127.0.0.1',
```

(continues on next page)

(continued from previous page)

```
    'port': 6379,
    'db': 0,
    'password': 'yourpassword'
}
```

CHAPTER 13

Templates

The templates mechanism is configured via the `TEMPLATE` variable in `${PACKAGE}/settings.py`. Templates are disabled by default.

13.1 Search Paths

The template files should live in a folder called `templates`, in your project's package directory. If you want them to be searched for in other folders, just add those paths to the `paths` template setting.

13.2 Basic Usage

To use the templating mechanism, just import it wherever you need it:

```
from colibris import template
```

To render a template file, simply call the `render` function and specify context as keyword arguments:

```
result = template.render('my_template.txt', var1='value1', var2=16)
```

To render a template from a string, call the `render_string` function:

```
result = template.render_string('Variable var1 is {{ var1 }} and var2 is {{ var2 }}.',
↪ var1='value1', var2=16)
```

13.3 Rendering HTML

The following example will render an HTML template file from a view:

```
from colibris.shortcuts import html_response_template

def index(request):
    return html_response_template('index.html', var1='value1')
```

13.4 Jinja2 Backend

Make sure to have the `jinja2` python package installed.

In `${PACKAGE}/settings.py`, set:

```
TEMPLATE = {
    'backend': 'colibris.template.jinja2.Jinja2Backend',
    'extensions': [...],
    'translations': 'gettext'
}
```

Field `extensions` is optional and represents a list of extensions to be used by the Jinja2 environment.

Field `translations` is optional and, if present, will enable `gettext`-based Jinja2 translations. Its value is the path to a python object that implements the `gettext` functions (such as the standard library `gettext`).

CHAPTER 14

Email

The email sending mechanism is controlled by the `EMAIL` variable in `${PACKAGE}/settings.py`. Emails are disabled by default.

14.1 Basic Usage

To send an email, just import the `email` package wherever you need it:

```
from colibris import email
```

Then create an email message:

```
msg = email.EmailMessage('My Subject', 'my body', to=['email@example.com'])
```

You can now send it:

```
email.send(msg)
```

Sending is done using the "fire and forget" way; don't expect any result or exceptions from this call. Any errors that might occur will be logged, though.

Make sure you configure your `default_from` value in your `EMAIL` setting, in `${PACKAGE}/settings.py`:

```
EMAIL = {
    'default_from': 'myservice@example.com',
    ...
}
```

14.2 Attachments

Attaching a file is as simple as calling the `attach()` method:

```
with open('/path/to/myfile.pdf', 'rb') as f:
    msg.attach('myfile.pdf', f.read())
```

14.3 HTML Content

Sending HTML content is achieved by specifying the `html` argument to `EmailMessage`:

```
msg = email.EmailMessage('My Subject', 'my text body', html='<p>My HTML content</p>',␣
↪to=['email@example.com'])
```

The HTML content acts as an alternative to the body and will be used by the mail readers that are capable to show it.

14.4 Console Backend

In `${PACKAGE}/settings.py`, set:

```
EMAIL = {
    'backend': 'colibris.email.console.ConsoleBackend'
}
```

You'll see the email content printed at standard output.

14.5 SMTP Backend

Make sure you have the `aiosmtplib` python package installed.

In `${PACKAGE}/settings.py`, set:

```
EMAIL = {
    'backend': 'colibris.email.smtp.SMTPBackend',
    'host': 'smtp.gmail.com',
    'port': 587,
    'use_tls': True,
    'username': 'user@gmail.com',
    'password': 'yourpassword'
}
```

CHAPTER 15

Offloading

Running time-consuming, blocking tasks can be done by using the `taskqueue` functionality in separate workers. The `TASK_QUEUE` variable in `${PACKAGE}/settings.py` configures the background running task mechanism. Background tasks are disabled by default.

15.1 Usage

To run a background task, import the `taskqueue` wherever you need it:

```
from colibris import taskqueue
```

Then run your time consuming task:

```
def time_consuming_task(arg1, arg2):
    time.sleep(10)

...

try:
    result = await taskqueue.execute(time_consuming_task, 'value1', arg2='value2',␣
↪timeout=20)

except Exception as e:
    handle_exception(e)
```

15.2 RQ Backend

Make sure to have the `rq` and `redis` python packages installed.

In `${PACKAGE}/settings.py`, set:

```
TASK_QUEUE = {
    'backend': 'colibris.taskqueue.rq.RQBackend',
    'host': '127.0.0.1',
    'port': 6379,
    'db': 0,
    'password': 'yourpassword',
    'poll_results_interval': 1
}
```

15.3 Background Worker

To actually execute the queued background tasks, you'll need to spawn at least one worker:

```
./${PACKAGE}/manage.py runworker
```

CHAPTER 16

Health

You can (and should) implement your project-specific health check function by exposing the get_health function in app.py:

```
def get_health():
    if not persist.connectivity_check():
        raise app.HealthException('database connectivity check failed')

    return 'healthy'
```

CHAPTER 17

Testing

17.1 The `pytest` Framework

Colibris uses pytest to provide an integrated testing framework. All features, plugins and common practices available with `pytest` are available with Colibris as well.

17.2 Writing Tests

One should simply place tests in the `${PACKAGE}/tests` package. The `pytest` discovery mechanism will recursively look for modules starting with `test_` and will run any function that starts with `test_` or ends with `_test`.

The following functions, placed in a file named e.g. `test_health.py` will test the health status API endpoint:

```
async def test_health_check_healthy(web_app_client):
    resp = await web_app_client.get('/health')
    assert resp.status == 200

    j = await resp.json()
    assert j == 'healthy'


async def test_health_check_db_down(web_app_client):
    persist.get_database().drop()

    resp = await web_app_client.get('/health')
    assert resp.status == 500

    j = await resp.json()
    assert j['code'] == 'unhealthy'
```

17.3 Testing Utilities & Fixtures

`pytest` recommends building tests around *fixtures*. Colibris provides the `web_app_client` fixture which wraps the `aiohttp_client` fixture and allows simulating HTTP requests "directly", bypassing any network layer.

Other testing utilities can be found in the `utils` module:

```
from colibris.test import utils
```

For validating enveloped API responses, one can then use `utils.assert_is_envelope`:

```
resp = await web_app_client.get('/users')
assert resp.status == 200

j = await resp.json()
utils.assert_is_envelope(j, count=2)
```

17.4 The `test` Management Command

Running the tests is achieved by running the `test` management command:

```
./${PACKAGE}/manage.py test
```

Any arguments passed to this command are passed internally to `pytest`. Running `pytest` directly is not recommended and will probably fail.

17.5 Test Database

Colibris will use the `TEST_DATABASE` setting for persistence when running tests. In the absence of a field in this setting (which is by default), corresponding fields from `DATABASE` setting will be used, but `name` will be prefixed with `test_`.

The test database is created at the setup phase of each test and dropped at the teardown phase. Its structure is created using migration scripts. Populating it with data is the responsibility of the test writer.

17.6 The `fixtures` Module

The `./${PACKAGE}/tests/fixtures.py` module can be used to define project-specific testing fixtures as well as constants. Here's an example of a fixture that creates a test user in the database:

```
@pytest.fixture
def test_user():
    yield models.User.create(username='test_user', password='test_password',
                             first_name='Test', last_name='User',
                             email='testuser@example.com')
```

CHAPTER 18

Deployment

18.1 Dependencies and Pipfile

Add your dependencies to `Pipfile`:

```
nano Pipfile
```

For example, if you're using PostgreSQL, you may want to add:

```
[packages]
....
psycopg2-binary = "*"
...
```

18.1.1 Lock Down Versions

Lock your dependencies with their versions in `Pipfile.lock`:

```
pipenv lock
```

18.1.2 Install Dependencies

Install all of your project's dependencies:

```
pipenv sync
```

18.2 Using `setuptools`

The project's skeleton comes with a `${PACKAGE}/setup.py` file, effectively allowing your project to be packaged with `setuptools`.

To create a package of your project, run:

```
python setup.py sdist
```

You'll then find your packaged project at `dist/${PROJECT_NAME}-${VERSION}.tar.gz`. The version is automatically read from `${PACKAGE}/__init__.py`.

The provided setup file will create a console script having your project's main package name, that will basically do exactly what `manage.py` does.

One thing that is worth noting when using `setuptools` to deploy a project is that the `manage.py` file that used to be in your project's root folder will now live in the main package of your project.

18.3 Using Docker

If you want to deploy your service using Docker, you'll first need to edit `Dockerfile` and change it according to your needs:

```
nano Dockerfile
```

If you plan on using Docker Compose, you'll probably want to edit the `docker-compose.yml` file as well:

```
nano docker-compose.yml
```

18.3.1 Building Docker Image

You can manually build the image for your server like this:

```
docker build -t ${PROJECT_NAME}:${VERSION} .
```

If your project has multiple services (e.g. "server" and "worker"), you'll want to build and tag them separately:

```
docker build -t ${PROJECT_NAME}:server-${VERSION} --target server .
docker build -t ${PROJECT_NAME}:worker-${VERSION} --target worker .
```

18.3.2 Manually Run Container

You can run your container locally:

```
docker run -it ${PPROJECT_NAME}:${VERSION} -p 8888:8888
```

or, if you have multiple services:

```
docker run -it ${PPROJECT_NAME}:server-${VERSION} -p 8888:8888
docker run -it ${PPROJECT_NAME}:worker-${VERSION}
```

18.3.3 Using `docker-compose`

You can use `docker-compose` to build your images, instead of building them manually:

```
docker-compose build
```

To start your services, use:

```
docker-compose up
```

When you're done, shut it down by hitting `Ctrl-C`; then you can remove the containers:

```
docker-compose down
```

CHAPTER 19

Settings

19.1 The `settings` Module

Each project should have a `${PACKAGE}/settings.py` file, specifying settings that are particular for the project.

19.2 Settings Schemas

Settings that need to be specified at runtime and depend on the running environment can be supplied via environment variables.

Settings schemas are used to validate and adapt environment variables before being used as settings. You have to define your settings schemas that will handle the settings your project wants to collect from the environment.

The following example will use the `DEBUG`, `LISTEN` and `PORT` environment variables to configure the corresponding settings, when added at the end of your `${PACKAGE}/settings.py`:

```
from colibris.conf.schemas import SettingsSchema, fields

class GeneralSettingsSchema(SettingsSchema):
    DEBUG = fields.Boolean()
    LISTEN = fields.String()
    PORT = fields.Integer()

GeneralSettingsSchema().load_from_env(globals())
```

The `globals()` argument ensures overriding values defined in your `${PACKAGE}/settings.py` module.

Providing values for complex settings, such as `DATABASE` which is defined as a dictionary with parameters, can be done by specifying the name of the setting as variable prefix:

```
class DatabaseSettingsSchema(SettingsSchema):
    NAME = fields.String()
    HOST = fields.String()
```

(continues on next page)

(continued from previous page)

```
    PORT = fields.Integer()
    USERNAME = fields.String()
    PASSWORD = fields.String()

    class Meta:
        prefix = 'DATABASE_'
```

If your project tends to have many such settings schemas, it is recommended that you move them to an e.g. `${PACKAGE}/settingsshemas.py` module:

```
from colibris.conf.schemas import SettingsSchema, fields

class GeneralSettingsSchema(SettingsSchema):
    DEBUG = fields.Boolean()
    LISTEN = fields.String()
    PORT = fields.Integer()

class DatabaseSettingsSchema(SettingsSchema):
    NAME = fields.String()
    HOST = fields.String()
    PORT = fields.Integer()
    USERNAME = fields.String()
    PASSWORD = fields.String()

    class Meta:
        prefix = 'DATABASE_'

...

def load_from_env(target_settings):
    GeneralSettingsSchema().load_from_env(target_settings)
    DatabaseSettingsSchema().load_from_env(target_settings)
```

Then import it in `${PACKAGE}/settings.py` and simply call `load_from_env` at the end:

```
settingsschemas.load_from_env(globals())
```

Environment variables can be put together in a `.env` file that is located in the directory where you run your project from (usually the root folder of your project). This file should never be added to git.

If you want your variables to be part of your project's repository, you can add them to `${PACKAGE}/.env.default`, which should be added to git.

19.3 Available Settings

19.3.1 `API_DOCS_URL`

Controls the path where the API documentation is served. Defaults to `/api/docs`.

19.3.2 `AUTHENTICATION`

Configures the authentication backend. Should be defined as a dictionary with at least one entry, `backend`, representing the python path to the backend class. The rest of the entries are passed as arguments to the constructor.

19.3.3 AUTHORIZATION

Configures the authorization backend. Should be defined as a dictionary with at least one entry, `backend`, representing the python path to the backend class. The rest of the entries are passed as arguments to the constructor.

19.3.4 CACHE

Configures the cache backend. Should be defined as a dictionary with at least one entry, `backend`, representing the python path to the backend class. The rest of the entries are passed as arguments to the constructor.

19.3.5 DATABASE

Sets the project database. See this for examples of database URLs.

19.3.6 DEBUG

Enables or disables debugging. Defaults to `True`.

19.3.7 EMAIL

Configures the email backend. Should be defined as a dictionary with at least one entry, `backend`, representing the python path to the backend class. The rest of the entries are passed as arguments to the constructor.

19.3.8 LISTEN

Controls the interface(s) on which the server listens. Defaults to `'0.0.0.0'`.

19.3.9 LOGGING

Configures the logging mechanism. See logging.config for details.

19.3.10 LOGGING_OVERRIDES

Allows overriding parts of the logging configuration (for example silencing a library).

19.3.11 MAX_REQUEST_BODY_SIZE

Controls the maximum allowed size of a request body, in bytes. Defaults to `10MB`.

19.3.12 MIDDLEWARE

A list of all the middleware functions to be applied, in order, to each request/response. Defaults to:

```
[
    'colibris.middleware.handle_errors_json',
    'colibris.middleware.handle_auth',
    'colibris.middleware.handle_schema_validation'
]
```

19.3.13 PORT

Controls the server TCP listening port. Defaults to `8888`.

19.3.14 PROJECT_PACKAGE_DIR

Sets the path to the project directory. This setting is determined automatically and should not be changed.

19.3.15 PROJECT_PACKAGE_NAME

Sets the main project package name. This setting is determined automatically and should not be changed.

19.3.16 SECRET_KEY

Sets the project secret key that is used to create various tokens. Defaults to `None` and must be set explicitly.

19.3.17 TASKQUEUE

Configures the background tasks backend. Should be defined as a dictionary with at least one entry, `backend`, representing the python path to the backend class. The rest of the entries are passed as arguments to the constructor.

19.3.18 TEMPLATES

Configures the templates backend. Should be defined as a dictionary with at least one entry, `backend`, representing the python path to the backend class. The rest of the entries are passed as arguments to the constructor.

19.3.19 TEST_DATABASE

Similar to `DATABASE` but used when running tests. Missing fields are used from `DATABASE`. If `name` is not specified, `DATABASE['name']` with a `test_` prefix will be used.