Strapp

Release 0.2.1

unknown

Mar 21, 2023

CONTENTS:

CHAPTER

ONE

QUICKSTART

CHAPTER TWO

STRAPP

A mildly opinionated library used to boot**str**ap **app**s. Its primary use is to commonize (and test) the typical bespoke boilerplate most applications tend to reimplement to varying levels of sophistication.

All dependencies are intentionally optional, and exposed through extras in order to make opting into or out of specific strapp decisions and modules entirely optional.

2.1 Package Highlights

- SQLAlchemy
  - Session creation helper functions
    - Opt-in "dry run" session feature
  - Custom `declarative_base`
    - Opt-in created_at/updated_at columns
    - Opt-out `repr`able models
    - For type safety, utilize the mypy plugin

      ```
      [tool.mypy]
      plugins = 'strapp.sqlalchemy.mypy'
      ```
- Click
  - Context "Resolver"
- Flask
  - Non-decorator based route registration pattern (removes circular import issues)
  - Opt-in error handlers
  - Opt-in database handling
- Logging
  - Logging verbosity helper
- Sentry
  - Setup helper
  - Context helper
- Dramatiq

- Interface helpers (`configure`, `enqueue`, `get_results`)
- Declarative actors (`PreparedActor`)
- Optional middlewares: `SentryMiddleware`, `DatadogMiddleware`

2.2 Optional Integrations

Strapp is designed with integration with Configly and Setuplog, two of our other open sourced packages.

These are entirely optional, and explained at the relevant locations in the docs to which they apply.

CHAPTER

THREE

SQLALCHEMY

3.1 Model Base

The `declarative_base()` function, allows you to very concisely opt into two columns which are very commonly included on a majority of tables, `created_at` and :code`updated_at`.

3.1.1 Mypy

You may encounter typing-related issues such as:

```
error: Variable "models.Base" is not valid as a type
error: Invalid base class "Base"
```

Because of the dynamically created type, a mypy plugin is required. An example `pyproject.toml` file enabling the plugin would look like:

```
[tool.mypy]
plugins = ["strapp.sqlalchemy.mypy"]
```

3.2 Session

3.2.1 Configly Integration

The `config` argument accepts `URL` arguments as a `typing.Mapping`, in order to make usage with Configly more straightforward.

Applications might define their configuration using configly.

Listing 1: config.yml

```
...
database:
  drivername: postgesql+postgres
  host: <% ENV[DATABASE_HOST] %>
  username: <% ENV[DATABASE_USERNAME] %>
  ... etc ...
...
```

Then a typical strapp *Flask* app would construct a config instance once before app startup, and use that to hook up their database to the flask app.

Listing 2: wsgi.py

```
from configly import Config
from strapp.flask import callback_factory, sqlalchemy_database

config = Configly.from_yaml('config.yml')
callback = callback_factory(sqlalchemy_database, config.database)
app = create_app(callbacks=[callback])
```

And a typical strapp *Click* app, might use the resolver and produce the engine that way.

Listing 3: cli.py

```
from strapp.click import Resolver
from configly import Config

def config():
    return Config.from_yaml("config.yml")

 def postgres(config):
     return strapp.sqlalchemy.create_session(config.database)

resolver = Resolver(config=config, postgres=postgres)

@resolver.group()
def command(postgres):
    ...
```

In neither case are you required to actually use Configly, but it is intentionally easy to do!

CHAPTER

FOUR

CLICK

The CLI applications we write usually tend to follow the same pattern:

- All commands require logging/sentry to have been initialized
- All commands require being wrapped to handle uncaught exceptions
- All commands require one or more of: config, one or more database connections, or one or more API clients of some sort.
- Often we want some way to manage verbosity
- Often we want some way to run idempotent, equivalents of commands, without committing any changes they might make.

The mechanism strapp exposes to facilitate these requirements and is the `Resolver`.

The two primary goals were:

- Enable the production of the various objects cli command invocations might require lazily, such that any command which did not require i.e. config, did not load config.
- Reduce the boilerplate required to either inject or construct those objects.

4.1 Resolver

Again note: if at any time, the patterns expected by Strapp dont work in a given situation, resolver methods always return *Click*-native primitives which can used normally, using normal click patterns.

With that being said, a typical click project using Strapp tends to look like so:

```
pyproject.toml / setup.py
src/
    projectname/
        cli/
            __init__.py
            base.py
            commandset1.py
            commandset2.py
         ... the rest of the project
```

We then use the `pyproject.toml`/`setup.py` to produce an entrypoint script.

Listing 1: pyproject.toml

```toml
[tool.poetry.scripts]
projectname = "projectname.cli:run"
```

In order to avoid circular imports when making use of the resolver in dependent subcommands, we imperitively add the commands to the base cli, after everything has been constructed.

Listing 2: __init__.py

```python
# flake8: noqa
from platform_actions.cli import base, commandset1, commandset2

base.cli.add_command(commandset1.commandset1)
base.cli.add_command(commandset2.commandset2)


def run():
    base.cli()
```

In `base.py`, we produce callables for all the resolvable resources, and instantiate the resolver.

Listing 3: base.py

```python
import click
import strapp.click
import strapp.sqlalchemy
import projectname
from configly import Config

def config():
    return Config.from_yaml("config.yml")

def api_client(config):
    return projectname.api_client.APIClient(config.api_client)

def postgres(config, dry_run):
    return strapp.sqlalchemy.create_session(config.postgres, dry_run=dry_run)

def redshift(config, dry_run):
    return strapp.sqlalchemy.create_session(config.redshift, dry_run=dry_run)

resolver = strapp.click.Resolver(
    config=config,
    postgres=postgres,
    redshift=redshift,
    api_client=api_client,
)

@resolver.group()
@click.option("--dry-run", is_flag=True)
@click.option("-v", "--verbose", count=True, default=0)
def cli(config: Config, dry_run, verbose):
    resolver.register_values(dry_run=dry_run, verbosity=verbose)
```

Optionally, this `cli` base group is the ideal spot to integrate with *Logging*.

And finally, commandset1/2 can be structured however they please. We tend to follow a pattern like:

Listing 4: commandset1.py

```python
import click
from projectname.cli.base import resolver
import projectname

@resolver.group()
def commandset1():
    pass

@resolver.command(commandset1, help='subcommand')
@click.option('--some-option')
def subcommand(postgres, redshift, api_client):
    projectname.do_something(postgres, redshift, api_client)
```

While it doesn't make a difference from a Strapp perspective, keeping a strict barrier between the click cli structure and the actual code which performs the actions of the cli tends to make testing much easier, tests just need to produce test-stubs for the arguments rather than needing to interact with click's testing facilities.

4.2 Testing

We also include a testing module to reduce to boilerplate associated with testing cli commands.

CHAPTER

FIVE

FLASK

A key annoyance with how flask typically recommends your applications be set up is that things like your `app` instance, plugins, database handles, etc tend to be module-level attributes and it's tempting or even encouraged to import and use them in code elsewhere (which often leads to gnarly circular import issues).

The pattern that Strapp tries to encourage, encapsulates as much of that as possible, so that the "only" (read easy/obvious) way to do things avoids these problems entirely.

5.1 Setup

A typical `ls` of a project directory might look something like

```
app.py (generally, either this or __main__.py, and not both)
src/
    project/
        __main__.py
        routes.py
        errors.py
        views/
            ...
        ...
```

All app setup, and references to things like plugins, and config are encouraged to exist in either something like the `app.py` above, or the `__main__.py`. The idea being that its location encourages you to **not** try to import it.

There your file contents might look like so:

Listing 1: app.py or __main__.py

```
from configly import Config
from setuplog import setup_logging
from strapp.flask import create_app, callback_factory, default_error_handlers,␣
↪handler_exception, sqlalchemy_database

from project.routes import routes
from project.errors import CustomErrorType

config = Config.from_yaml('config.yml')

setup_logging(**config.logging.to_dict())

app = create_app(
    routes,
```

(continues on next page)

(continued from previous page)

```
    config=config.flask,
    error_handlers=[
        *default_error_handlers,
        handler_exception(CustomErrorType),
    ],
    plugins=[
        # FlaskCORS(),
        # FlaskWhateverPlugin(),
    ]
    callbacks=[
        callback_factory(sqlalchemy_database, config.database)
    ),
)
```

A couple of notes:

- As we'll see below, routes can be directly imported because the `routes` argument does not require a reference to `app` (as it would normally, though, you **can** always put this somewhere importable and do route setup in a more typical way).
- the intent is to centralize and pre-instantiate any shared objects which views might otherwise try to import, and make them available to views through means other than direct import.
- This callback mechanism, (along with `strapp.flask.inject_db()` below), are recommended rather than using e.g. FlaskSQLAlchemy (which encourages and/or requires a circular dependence of your models on a flask-specific plugin), though once again you can use whatever you'd like.

5.2 Routes

Listing 2: routes.py

```python
from strapp.flask import Route

from project.views import x, bar

routes = [
    ('GET', '/foo', x.get_x),
    Route.to('POST', '/bar', y.create_bar, endpoint='create_bar'),
    dict(method='GET', path='/foo', view=x.get_x),
]
```

We try to be as flexible as possible in allowing the routes to be defined concisely. Ultimately, all the arguments boil down to the arguments sent into `flask.Flask.route()`, however the actual reference (and attachment) to the app is delayed until the call to `strapp.flask.create_app()`.

5.3 Views

Finally, for defining actual view functions, there are additional decorators which can be used to simplify a typical (usually json) route.

CHAPTER

SIX

LOGGING

strapp.logging.**package_verbosity_factory**(**definitions*)

Describe per-package verbosity.

Each *definitions* item should be an iterable which describes the progression of log level at each level of verbosity.

Examples

This states that: *urllib3* should start at INFO logs by default, then only increase to DEBUG at 3 verbosity (i.e. -vvv). And that *sqlalchemy* should start at WARNING, then INFO (at -v), and DEBUG (at -vv, -vvv, etc)

```
>>> package_verbosity = package_verbosity_factory(
...     ("urllib3", logging.INFO, logging.INFO, logging.INFO, logging.DEBUG),
...     ("sqlalchemy", logging.WARNING, logging.INFO, logging.DEBUG),
... )
>>> package_verbosity(0)
{'urllib3': 20, 'sqlalchemy': 30}
>>> package_verbosity(1)
{'urllib3': 20, 'sqlalchemy': 20}
>>> package_verbosity(2)
{'urllib3': 20, 'sqlalchemy': 10}
>>> package_verbosity(3)
{'urllib3': 10, 'sqlalchemy': 10}
```

Omitting the log level progression entirely isn't really an intended usecase, but the returned callable accepts a *default* arg, which can be used to control the behavior in this case. It defaults to *logging.INFO*

```
>>> package_verbosity = package_verbosity_factory(["urllib3"])
>>> package_verbosity(0)
{'urllib3': 20}
```

```
>>> package_verbosity(0, default=logging.DEBUG)
{'urllib3': 10}
```

Returns A callable which, given the desired *verbosity*, returns a mapping of package to log level.

This is **most** useful when paired with setuplog or loguru.

- Setuplog's `setup_logging()` accepts a `log_level_overrides` argument who's format matches the output of this function.
- Loguru's `loguru.add()` function accepts a `filter` argument which who's format is compatible with the output of this function.

For use in applications, it's generally useful to call this as early as it is possible to have the level of verbosity you want.

For cli applications, this will tend to be inside the root cli group, where you accept the verbosity level.

```
import logging
import strapp.logging
from setuplog import setup_logging

package_verbosity = strapp.logging.package_verbosity_factory(
    ("urllib3", logging.INFO, logging.INFO, logging.INFO, logging.DEBUG),
    ("sqlalchemy.engine", logging.WARNING, logging.WARNING, logging.INFO, logging.
↪DEBUG),
    ("docker", logging.INFO, logging.INFO, logging.INFO, logging.DEBUG),
)

@click.group()
@click.option('--verbose', count=True, default=0)
def cli(verbose):
    setup_logging(
        config.logging.level, log_level_overrides=package_verbosity(verbose),
    )
```

For flask apps, this tends to be even earlier, i.e. as soon as you load the config.

CHAPTER

SEVEN

SENTRY

CHAPTER EIGHT

DATADOG

CHAPTER NINE

CONTRIBUTING

9.1 Prerequisites

If you are not already familiar with Poetry, this is a poetry project, so you'll need this!

9.2 Getting Setup

See the `Makefile` for common commands, but for some basic setup:

```
# Installs the package with all the extras
make install
```

And you'll want to make sure you can run the tests and linters successfully:

```
# Runs CI-level tests, with coverage reports
make test lint
```

9.3 Need help

Submit an issue!

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX