Bors Service Integrator Documentation

Release 0.3.5

Bobby

Jun 25, 2019

Contents

Contents:

CHAPTER 1

Bors

A highly flexible and extensible service integration framework for scraping the web or consuming APIs.

CHAPTER 2

Usage

1. Create your model based on the data you expect to incorporate.
2. Decide on what you want to do with your data, and add it.
3. Create or use an existing API integration library.
4. Create your root application to tie it all together.

2.1 Object Model

We use marshmallow for the underlying object schema definitions. Here's an example model:

```
from marshmallow import Schema, fields

class NewsItemSchema(Schema):
    """News item"""
    id = f.Str(required=True)
    url = f.Str(required=True)
    title = f.Str(required=True)
    pubDate = f.Str(required=True)
    timestamp = f.Str(required=True)
    feed_id = f.Int(required=True)
    published_date = f.Str(required=True)
    feed_name = f.Str(required=True)
    feed_url = f.Str(required=True)
    feed_enabled = f.Int(required=True)
    feed_description = f.Str(required=True)
    url_field = f.Str(required=True)
    title_field = f.Str(required=True)
    date_field = f.Str(required=True)
    feed_image = f.Str(required=True)
```

See the `marshmallow` docs for more information.

2.2 Middleware Strategies

Middleware API is implemented in the form of strategies and follows this basic layout:

```
"""
Simple context display strategy
"""

from bors.app.strategy import IStrategy


class Print(IStrategy):
    """Print Strategy implementation"""
    def bind(self, context):
        """
        Bind the strategy to the middleware pipeline,
        returning the context
        """
        print(f"""PrintStrategy: {context}""")

        # just a pass-through
        return context
```

The important things to note here: * We're inheriting from `IStrategy`. * We're implementing a `bind` method. * The bind method receives, potentially augments, and then returns the `context`.

2.3 API Integration

2.3.1 Request Schema

Because our API is simple, we're going to use this as-is.

```
from bors.generics.request import RequestSchema
```

2.3.2 Response Schema

Our API sends us data in the following format:

```
{
    "data": ...,
    "status": "OK"
}
```

For this, we'll need to supplement a bit, removing the root fields and returning the `data` value:

```
from marshmallow import fields
from bors.generics.request import ResponseSchema


class MyAPIResponseSchema(ResponseSchema):
    """Schema defining how the API will respond"""
    status = fields.Str()
    def get_result(self, data):
```

(continues on next page)

(continued from previous page)

```
        """Return the actual result data"""
        return data.get("data", "")

    class Meta:
        """Add 'data' field"""
        strict = True
        additional = ("data",)
```

2.3.3 API Class

```
from bors.api.requestor import Req


class MyAPI(LoggerMixin):
    name = "my_api"
    def __init__(self, context):
        self.create_logger()

        self.request_schema = RequestSchema
        self.result_schema = MyAPIResponseSchema
        self.context = context

        self.req = Req("http://some.api.endpoint/v1", payload, self.log)

        # We don't need to deal directly with requests, so we pass them through
        self.call = self.req.call

    def shutdown(self):
        """Perform last-minute stuff"""
        pass
```

Here we use the built-in `Req` class to issue requests to the API, we assign the `request_schema` and `result_schema` to classes in our object, and we set the `name`, `context`, and `call` attributes. The results passed through on the API are referencable from within the middleware context under the key `my_api`.

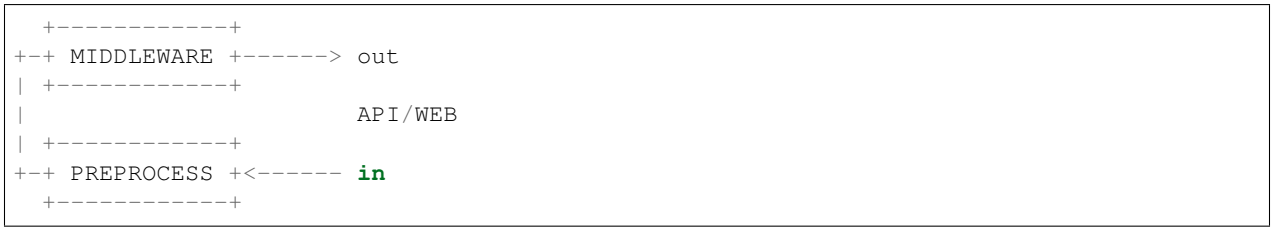
2.3.4 Pulling it all together

```
from bors.app.builder import AppBuilder
from bors.app.strategy import Strategy


def main():
    strat = Strategy(Print())
    app = AppBuilder([MyAPI], strat)
    app.run()

if __name__ == "__main__":
    main()
```

Here, we set as many strategies and API's as we want, then create and run the `app`.

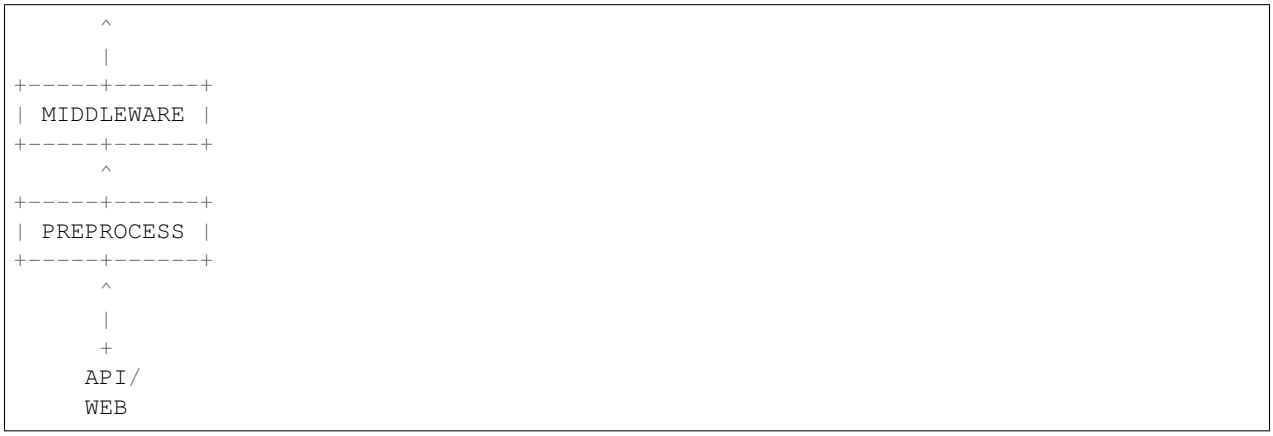
CHAPTER 3

Architecture



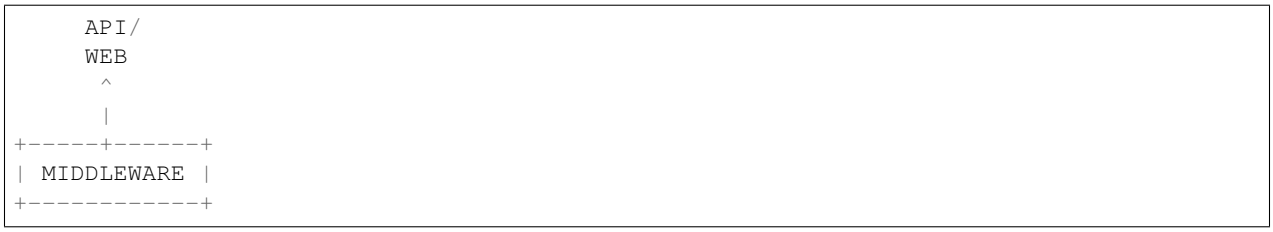
At its most basic level, a `bors` integrator engages with an integration library (API) passing incoming data through a prepocessor to generate and validate incoming objects, then passes that data through middlewares. Outgoing interactions are initiated from within a middleware and passed directly to an API, allowing easily for request/response type behavior in addition to observe and react.

3.1 Ingesting Data



Ingested data provokes calls along the pipeline.

3.2 Outgoing Data



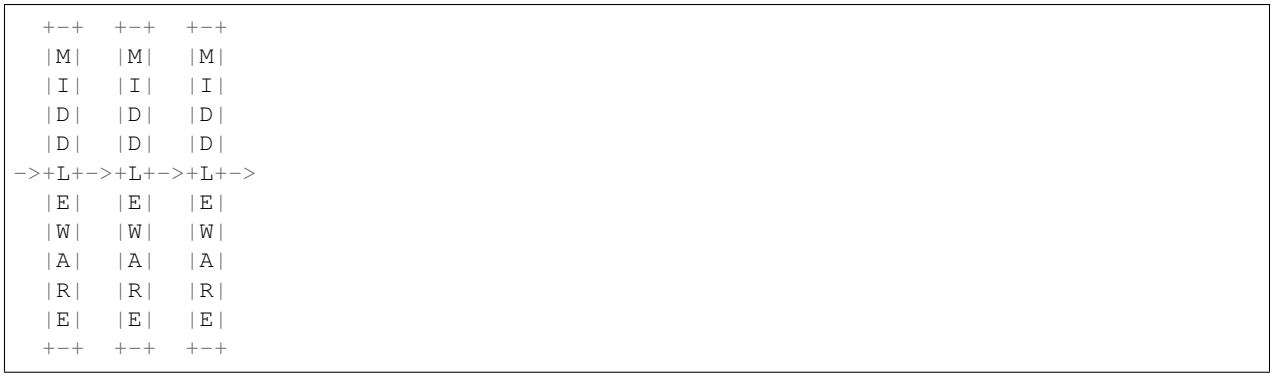
Enacted events stimulate API or web actions.

3.3 Preprocessing

Preprocessing is nothing more than an object-ization of the incoming data. This provides two benefits: 1. Data can be generalized across API interfaces. 2. Data structure can be validated and enforced.

3.4 Middlewares

Middlewares allow for a data processing pipeline to pass data through.



With this model, we gain a lot of flexibility in the behavior of our integration. Middleware is up to the developer to create, and can be any of the following:

- Data post-processing, filtering, aggregation, or augmentation
- External integrations and interfaces
- Stimulate an API/web transaction from external actors or time-based criteria
- Hooks and callbacks

CHAPTER 4

Installation

At the command line:

```
$ easy_install bors
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv bors
$ pip install bors
```

CHAPTER 5

Usage

To use Bors Service Integrator in a project:

```
import bors
```

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/karma0/bors/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

6.1.4 Write Documentation

Bors Service Integrator could always use more documentation, whether as part of the official Bors Service Integrator docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/karma0/bors/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *bors* for local development.

1. Fork the *bors* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/bors.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv bors
$ cd bors/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 bors tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/karma0/bors/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_bors
```

CHAPTER 7

Credits

7.1 Development Lead

- Bobby <karma0@gmail.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

History

CHAPTER 9

0.3.5 (2018-06-26)

- Added badges.
- Cleaned up deps.

CHAPTER 10

0.3.4 (2018-06-26)

- Pruned and upgraded all dependencies.

CHAPTER 11

0.3.3 (2018-06-26)

- Cleaned up tox.ini

CHAPTER 12

0.3.2 (2018-06-26)

- Setup example.py script for others to use.

CHAPTER 13

0.3.1 (2018-06-26)

- Added several unit tests, building out some of the framework.

CHAPTER 14

0.3.0 (2018-06-25)

- Reboot packaging using cookiecutter-pypackage.

CHAPTER 15

0.2.0 (2018-05-17)

- Initialize the repository, breaking it out from nombot.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search