py-fromjson Documentation

Release 0.1.0

Dmitry Bogatov

Aug 09, 2020

Contents

1 Rationale **3**
1.1 Contents: 5
1.2 Feedback 7

pypi package ?.?.?

build unknown

Parse dataclasses from unstructured JSON in Python3.

CHAPTER 1

Rationale

The most common format for client to send data to server is JSON, and all Python web framework way to access request body as nested dictionaries of primitive values. Problem is that programmer is supposed to thorously validate input data, or bad input can cause request handler to throw exception and return http code 500 instead of 400.

Let us consider quite typical setup – Sanic web framework and aiopg library and very trivial example of application that returns author of the book identified by title and publishment year, year is optional.

```
@app.route("/book", methods=["GET"])
def return_book_author(request):
    title = request.args["title"]
    year = request.args.get("year")

    with (await request.app.pool.cursor()) as cursor:
        cursor.execute("""
          SELECT author FROM books
          WHERE title = %s AND year = COALESCE(%s, year)
        """, [title, year])
        authors = [row[0] for row in (await cursor.fetchall())]
   return json(authors)
```

A lot of things can go wrong with this code. Client may not have send `title` parameter, or may have sent it of wrong type. Parameter `year` can also be of wrong type. It is not disaster, but database will throw error, which in turn will result in http code 500. Every time frontend developers see 5xx, they rightfully assume that it is not their fault, and bug backend developer. To avoid such unfortunate consequences, types must be thorously checked:

```
@app.route("/book", methods=["GET"])
def return_book_author(request):
    title = request.args.get("title")
    year = request.args.get("year")

    if type(title) is not str:
        return text("Bad type of parameter `title', must be string", 400)
    if year is not None and type(year) is not int:
        return text("Bad type of parameter `year', must be int or absent", 400)
```

(continues on next page)

(continued from previous page)

```
    with (await request.app.pool.cursor()) as cursor:
        cursor.execute("""
          SELECT author FROM books
          WHERE title = %s AND year = COALESCE(%s, year)
        """, [title, year])
        authors = [row[0] for row in (await cursor.fetchall())]
   return json(authors)
```

It happens to work, but does not scale. With more parameters to check and more time pressure applied, programmer will skip some checks and will assume happy path. It will backfire, no exceptions.

This library provides much simpler way to do required parsing and checking:

```
from dataclasses import dataclass
import typing
import fromjson

@fromjson.derive
@dataclass
class BookRequest:
   title: str
   year: typing.Optional[int]

@app.route("/book", methods=["GET"])
def return_book_author(request):
    try:
        book = BookRequest.fromjson(request.args)
    except ValueError as e:
      return text(str(e), 400)

    with (await request.app.pool.cursor()) as cursor:
        cursor.execute("""
          SELECT author FROM books
          WHERE title = %s AND year = COALESCE(%s, year)
        """, [book.title, book.year])
        authors = [row[0] for row in (await cursor.fetchall())]
   return json(authors)
```

That's it. If `fromjson` method did not throw exception, code after it can be sure that `title` field is string, and `year` field is either integer or `None`.

This library is heavily inspired by Haskell type system and following Haskell libraries in particular:

- aeson
- refined
- servant

If you have choice, you should just use haskell. It you, like me, is stuck with Python, read on to learn how you can use `py-fromjson` library to make your code more correct and safe.

https://hackage.haskell.org/package/aeson-1.5.3.0/docs/Data-Aeson.html#t:FromJSON

1.1 Contents:

1.1.1 Installation

At the command line either via easy_install or pip:

```
$ easy_install py-fromjson
$ pip install py-fromjson
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv py-fromjson
$ pip install py-fromjson
```

1.1.2 Usage

To use py-fromjson in a project:

```
import fromjson
```

1.1.3 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:

Types of Contributions

Report Bugs

Report bugs at https://github.com/kaction/py-fromjson/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.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/kaction/py-fromjson/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 :)

Get Started!

Ready to contribute? Here's how to set up *py-fromjson* for local development.

1. Fork the *py-fromjson* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/py-fromjson.git
```

3. Create a branch for local development:

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

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass style and unit tests, including testing other Python versions with tox:

```
$ tox
```

To get tox, just pip install it.

5. 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
```

6. Submit a pull request through the GitHub website.

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, and 3.3, and for PyPy. Check https://travis-ci.org/kaction/py-fromjson under pull requests for active pull requests or run the `tox` command and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test test/test_py-fromjson.py
```

1.1.4 Credits

Development Lead

- Dmitry Bogatov <KAction@disroot.org>

Contributors

None yet. Why not be the first?

1.1.5 History

0.1.0 (2020-08-02)

- First release on PyPI.

1.2 Feedback

If you have any suggestions or questions about **py-fromjson** feel free to email me at KAction@disroot.org.

If you encounter any errors or problems with **py-fromjson**, please let me know! Open an Issue at the GitHub http://github.com/kaction/py-fromjson main repository.