Software
Sustainability
Institute

SSI/HEPData - Research Software Healthcheck

Project Title	SSI/HEPData
Authorship	James Graham
Document Version	v1.0
Date	2020-03-25

1 Introduction

1.1 Overview

This evaluation has looked at several aspects of the HEPData software: its dependencies, support for the development process, developer documentation and the code itself.

The project appears in general to be well managed, structured and developed, with only one major concern - lack of support for Python3. There are a few areas where moderate improvement may be made, each of which is highlighted at the end of each section.

1.2 Recommendations for First Steps

The areas where the most improvement stands to be made are:

- Complete update to Python3
    - Update dependencies
- Expand developer documentation - particularly system architecture
- Modularisation and documentation of JavaScript component

2 Dependencies

This area focuses on the dependencies of the software being evaluated - are they a good fit for the project requirements and are they managed well?

HEPData is built upon the Invenio Framework[1], an open source data repository framework originally developed at CERN and used by a number of other large data repositories such as Zenodo. Invenio appears to be well supported, with an active commit history and issue tracker. However, the version of Invenio used by HEPData is pinned at 3.0.0a14 - indicating an unsupported alpha version. Furthermore, the 3.0.x series reached End of Life at the end of 2019 [2]. The current version, 3.2.1 will be supported until at least the end of 2020 and supports both Python 2.7 and 3.6 so should not interfere with Python migration efforts.

For the recommended deployment process, Python dependencies are managed using a virtual environment with the majority of packages in the requirements.txt file having a pinned version to support reproducibility.

JavaScript dependencies were installed via npm, but were not specified in a file, pinned to a particular version, or installed within a local environment. Here I would recommend making the management of JavaScript dependencies as similar as possible to how the Python dependencies are managed. Using pinned versions in a local environment would help to greatly reduce the chance of version conflicts for developers working on multiple projects - particularly for external developers wanting to contribute to the project.

The install documentation mentions problems with running the npm install in local mode. If it is not possible to resolve the issue and allow packages to be installed in local mode, I would ideally like to see here an explanation of the problem that was encountered. By presenting an explanation here, the chances of another developer recognising a solution would be increased.

Observations and suggestions:

- Python dependencies well managed, though in some cases unsupported versions are used
- Treating JavaScript dependencies in the same way as Python would help to reduce potential for conflict

[1] https://inveniosoftware.org/
[2] https://invenio.readthedocs.io/en/latest/releases/maintenance-policy.html#end-of-life-releases

3 Development Process

This area focuses on the use of software engineering tools and processes within the project - how is change tracked and monitored?

The source code is publicly visible and appropriately licensed on GitHub[3]. The issue tracker is well used, with categorisation of issues by featureset affected. The code is protected from modification by locking of the master branch to direct contributions, though it appears that project owners are able to push directly to the master branch.

Continuous Integration is in place using Travis CI[4] with a good number of tests, providing relatively high coverage of the source[5]. This will be very useful during the migration to Python3 - here I would recommend adding a Python3 configuration to the CI and using the existing tests to ensure that the project remains compatible with both Python2 and 3 until it can be reasonably expected that all deployments have been migrated to Python3.

Commit messages and issues are consistently descriptive, providing enough detail to fully understand the scope of the changes. A very minor point here is that there are two different styles of commit message in use, so standardising on one style would provide a small benefit in ease of use when looking for a particular change. One of the styles in use is very similar to Conventional Commits[6] - a relatively recently standardised format which aims to concisely define both the scope and consequences of a changeset, making it easier for developers and automated tools to reason about changes. This would be the format I would recommend if standardising on one.

Observations and suggestions:

- The development process appears to be well managed, there are no major suggestions for improvement
- Aiming for greater consistency in style between developers may provide a small benefit, but this seems very low priority

[3] https://github.com/HEPData/hepdata
[4] https://travis-ci.org/github/HEPData/hepdata/branches
[5] https://coveralls.io/github/HEPData/hepdata
[6] https://www.conventionalcommits.org/en/v1.0.0-beta.4/

4 Developer Documentation

This area focuses on the information available to help introduce external developers to the project - is the structure and purpose of each component sufficiently well described?

Developer documentation is compiled and deployed automatically using ReadTheDocs[7]. Python docstrings are used relatively consistently to provide detailed API-level developer documentation. Each documentation page describing a package (e.g. Converter[8]), begins with a table briefly describing the modules contained within it. These tables are useful, but could be expanded a little to provide more context.

Additionally, it would be useful to see a similar table at the top level, briefly describing each of the packages. Again, this would help make it easier for a new or external developer to contribute to the project - for example, if I contribute code for converting Records to a format compatible with another source, should I look in the Converter or Records package? These additions to the documentation could be placed in the Python source as package and module level docstrings (which may contain ReStructuredText), so that they are visible when making changes to the code. This would ensure that they do not become out of date when substantial changes are made, but also are close to hand when they are most useful.

For the same reason, it is often beneficial to include a high level architecture diagram in the developer documentation. This does not need to be comprehensive, but could just define the core components and the relationships between them.

Finally, I would suggest a pass over the existing docstrings to add a bit more detail to some, where this is lacking. Many of the docstrings are sufficiently complete with detailed descriptions of the aim, method, input and output of the code, but some are missing important detail.

Observations and suggestions:

- API level developer documentation is generally good and appropriately managed, but is sometimes incomplete
- Architecture level documentation would help to ease the introduction of external developers to the project and would make it easier to reason about the structure of the software

[7] https://hepdata.readthedocs.io/en/latest/?badge=latest
[8] https://hepdata.readthedocs.io/en/latest/modules/converter.html

5 Code

This area focuses on the code itself - is it well structured and making good use of current best practices?

5.1 Python

The Python component of HEPData is consistently cleanly structured and formatted, resulting in code which is easy to read and understand. The use of docstrings further aids readability.

The major issue here is that the project does yet support Python3. This is a known issue and work is in progress to address it, but it is included here to stress its importance. Python2 has been unsupported since the beginning of 2020, so is no longer receiving security updates, unless support has been contracted with a third party. As a consequence of this, Python2 is gradually being removed from many package managers and will be increasingly difficult to deploy.

The only other comment on the Python component of HEPData is the use of 'bare except' clauses - the use of *try-except* without specifying a type of exception to be captured. This is usually overly general and can result in the software attempting to recover from an unrecoverable failure. Its use in HEPData is usually to rollback a database transaction, which is one of the few valid uses. However, it is still generally prefered to catch specific recoverable exceptions and handle them, with unrecoverable exceptions propagated up the stack to such a point as they are able to be caught and logged centrally. It is important that all exceptions representing a failure of the system to handle a set of conditions are logged and visible to administrators to aid in identifying any bugs / weaknesses which may have caused this to happen. Again, this is usually done in HEPData, but there were a number of cases where bare exceptions were caught and this was not logged.

5.2 JavaScript

The JavaScript component of HEPData is relatively cleanly organised and formatted, but is largely undocumented. I suspect the reason for this is that there is no one standard documentation format for JavaScript. In the absence of a single standard, the Google style guides are usually a good default. The Google JavaScript style guide recommends JSDoc[9] for inline documentation, for which Sphinx plugins exist[10] to add the JavaScript documentation to the existing online documentation.

Depending on the browsers (family and version) which need to be supported by the project, it may be beneficial to introduce some of the newer features of JavaScript (ES6) to make the existing code more robust. The most important newer additions to help ensure correctness are;

[9] https://google.github.io/styleguide/jsguide.html#jsdoc
[10] https://github.com/mozilla/sphinx-js

let[11] and *const*[12] instead of var for limiting the scope of variables, strict mode[13] to highlight several common potential errors or mistakes, and the Fetch API[14] as an alternative to JQuery AJAX with better support for error handling. To help with structure, modules[15] allow the same breakdown of JavaScript code as is possible with Python. The native module system is an alternative to tools such as Webpack[16] which may be simpler to implement, though is compatible only with more recent browsers.

When updating JavaScript to make use of features in newer standards, it is important to check that the proposed features are compatible with all of the browsers being targeted for support. For this, the website CanIUse[17] is a useful resource.

Observations and suggestions:

- Python2 is no longer supported, so completing the migration to Python3 should be the first priority
- Consider how exceptions are handled within the Python component - is there sufficient information available when investigating failures of the production system?
- The JavaScript component is generally well organised, but largely undocumented - JSDoc would be a sensible choice of documentation style in the absence of any other preference
- The JavaScript component may also benefit from some features added in ES6 to improve robustness and structure

[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
[13] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
[14] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
[16] https://webpack.js.org/
[17] https://caniuse.com/