ANACONDA.

Intro to Conda Recipe Manager

Schuyler Martin, Senior Software Engineer

June 14th, 2024

Open Source Project. © 2023 Anaconda

Conda Packaging

A *very* brief overview

- `conda` provides tools to easily consume Python packages
  - A *package* is a bundle of built software components
- We build these packages with `conda-build` using *recipe files*
  - A *recipe* is a set of instructions and metadata that tells a build system how to package software and what other packages the project depends on
- `conda` uses a partially documented and custom recipe file format

Conda Packaging

A *very* brief overview

- Unfortunately:
  - The current recipe format is very difficult to work with and has a lot of issues
    - (More on that later)
- Fortunately:
  - The Conda Community has worked very hard the last few years and has proposed a new recipe format that addresses a lot of the problems.
  - CEP-13 & CEP-14 define the "core" of the new format.
    - And there are more CEPs under discussion right now!

Open Source Project. © 2024 Anaconda

Comparing the two formats

There are reasons for the switch

V0 Recipe Format (Pre CEP-13/14)

- Is not valid YAML
- Not standardized or defined
- Incredibly difficult to parse, requiring multiple passes
    - Comments have meaning (selectors)
    - Supports complex JINJA constructs, like loops and if statements
- Difficult to edit files programmatically
- Allows arbitrary execution of Python code
- Recipes are stored in `recipe/meta.yaml`

V1 Recipe Format (CEP-13/14+)

- Is valid YAML
- Standardized - enforced and clearly defined by CEPs
- Can be parsed/schema-validated in one pass
- JINJA notation is limited and $-escaped
- Selectors have been replaced with parseable YAML if/then-objects (and only allowed under certain scenarios)
- Contains a `schema_version` value for future-proofing/easy upgrading
- Recipes are stored in `recipe/recipe.yaml`

Example: V0 Format for types-toml-feedstock

```yaml
{% set name = "types-toml" %}
{% set version = "0.10.8.6" %}

package:
  name: {{ name|lower }}
  version: {{ version }}

source:
  url: https://pypi.io/packages/source/{{ name[0] }}/{{ name }}/types-toml-{{ version }}.tar.gz
  sha256: 6d3ac79e36c9ee593c5d4fb33a50cca0e3adceb6ef5cff8b8e5aef67b4c4aaf2

build:
  number: 0
  skip: true  # [py<37]
  script: {{ PYTHON }} -m pip install . -vv --no-deps --no-build-isolation

requirements:
  host:
    - setuptools
    - wheel
    - pip      You, 3 months ago • Initial migration work …
    - python
  run:
    - python

test:
  imports:
    - types
  requires:
    - pip
  commands:
    - pip check
    - test -f $SP_DIR/toml-stubs/__init__.pyi  # [unix]

about:
  home: https://github.com/python/typeshed
  summary: Typing stubs for toml
  description: |
    This is a PEP 561 type stub package for the toml package.
    It can be used by type-checking tools like mypy, pyright,
    pytype, PyCharm, etc. to check code that uses toml.
  license: Apache-2.0 AND MIT
  license_file: LICENSE
  license_family: OTHER
  dev_url: https://github.com/python/typeshed
  doc_url: https://pypi.org/project/types-toml/

extra:
  recipe-maintainers:
    - fhoehle
    - conda-forge/mypy
```

Example: V1 Format for types-toml-feedstock

```yaml
schema_version: 1

context:
  name: types-toml
  version: 0.10.8.6

package:
  name: ${{ name|lower }}
  version: ${{ version }}

source:
  url: https://pypi.io/packages/source/${{ name[0] }}/${{ name }}/types-toml-${{ version }}.tar.gz
  sha256: 6d3ac79e36c9ee593c5d4fb33a50cca0e3adceb6ef5cff8b8e5aef67b4c4aaf2

build:
  number: 0
  skip:
    - py<37
  script: ${{ PYTHON }} -m pip install . -vv --no-deps --no-build-isolation

requirements:
  host:
    - setuptools
    - wheel
    - pip
    - python
  run:
    - python
```

```yaml
tests:
  - python:
      imports:
        - types
      pip_check: true
  - requirements:
      run:
        - pip
    script:      You, 3 months ago • Fixes bugs found by `types-toml` testing …
      - if: unix
        then: test -f $SP_DIR/toml-stubs/__init__.pyi

about:
  summary: Typing stubs for toml
  description: |
    This is a PEP 561 type stub package for the toml package.
    It can be used by type-checking tools like mypy, pyright,
    pytype, PyCharm, etc. to check code that uses toml.
  license: Apache-2.0 AND MIT
  license_file: LICENSE
  homepage: https://github.com/python/typeshed
  repository: https://github.com/python/typeshed
  documentation: https://pypi.org/project/types-toml/

extra:
  recipe-maintainers:
    - fhoehle
    - conda-forge/mypy
```

Why do we want to move to this big format?

What's the big deal?

- Logistics
  - The CEPs introducing the V1 format have been approved by the Conda Community
    - conda-forge WILL be moving to this format in the future
    - conda-forge's tooling will start using this format
  - Anaconda Recipes are often based on conda-forge recipes
    - If we don't address this now, our package builders will have to manually migrate recipes back to the old format or start from scratch when adding new packages
    - There will be inherit incompatibilities between conda-forge's ecosystem and what Anaconda does internally

Open Source Project. © 2024 Anaconda

Why do we want to move to this big format?

What's the big deal?

- Speed & Scale
  - The V0 recipe format is inherently slow to work with
    - Requires multiple passes to build a recipe
    - Format is not officially defined
      - Optimizations have a high risk of breaking conda-build
      - It is very difficult to edit files in an automated workstream (more on this later)
  - The V1 format will enable a significant improvement in our daily operations
    - It takes only 1 pass to read a recipe file
    - As the language is clearly defined, static dry run evaluations are now possible
    - Current estimates/experiments have shown at least a 10x speed improvement on simple builds
      - Time is money (and carbon emissions)
    - Since the format is pure YAML, developing automation around editing recipe files should be much easier

Conda Recipe Manager (CRM)

A monument to all our sins

- Conda Recipe Manager makes managing recipe files easy!
  - This project provides a number of tools that make it possible
  - The core of the project is a library that provides tools for editing recipe files
  - The project also provides a series of command line interfaces to access these tools and perform bulk operations

- History
  - Started as a module in percy and heavily used by the Anaconda Linter project to automatically edit V0 recipe files, reducing workloads on our package builders
    - Automate the easy stuff so we have more time for the hard stuff
  - As the conda community started to get more invested in the V1 format, it became clear that the recipe parser component could be used to help transition to the new format

Conda Recipe Manager (CRM)

A monument to all our sins

There are two major components of the CRM project:

- 1) The recipe parser
  - Reads in the keys (fields) and values and stores them in a parse tree
  - Selectors, comments, and JINJA variables are stored in several auxiliary data structures
  - No support for JINJA for loops or conditionals
    - Our estimates suggests <10% of all recipes use these more advanced constructs
  - Provides access and mutation functionality for fields, selectors, and variables
  - Patching recipe files uses a JSON Patch like syntax that is *nearly* RFC-9602 compliant
  - Also provides tools for:
    - Debugging the parse tree and auxiliary data structures
    - Patching on paths that match a regular expression
    - Providing a diff against the previous recipe state

Conda Recipe Manager (CRM)

A monument to all our sins

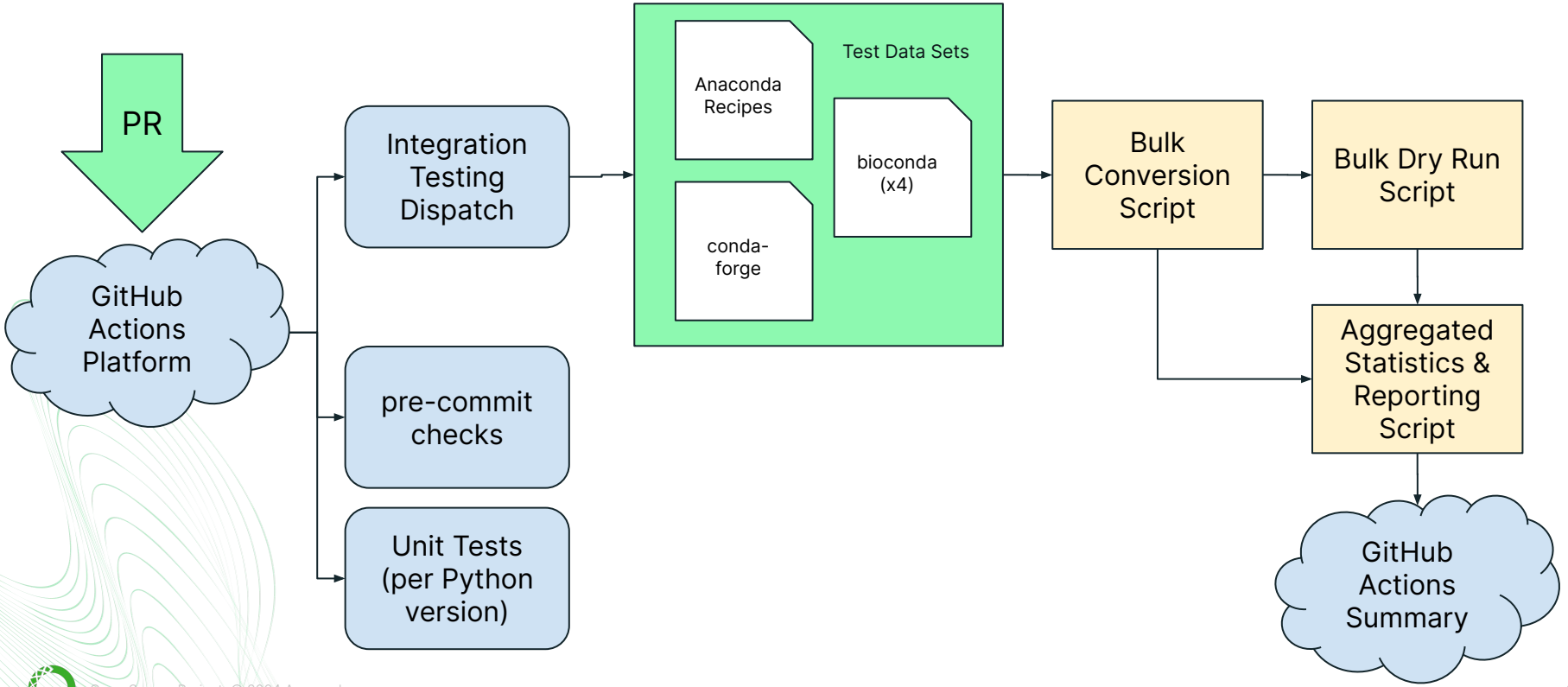
There are two major components of the CRM project:

- 2) The recipe converter
  - Extends the parser class
  - Contains two phases:
    - The preprocessor phase
      - Corrects common mistakes, difficult-to-parse scenarios, and other oddities
    - The rendering/converting phase
      - Performs JSON patch operations (and other parse tree manipulations) to transform the old recipe parse tree to the new format
      - Returns a string that is the raw text of the new recipe file
      - Failed operations and other operations are logged into a custom logger, to be used at the discretion of the calling program

Challenges

- How do we parse a file so complex and ill-defined?
  - Computer Science and patience!
- How do we ensure progress with such a complex system?
  - Automation!
    - Every commit must
      - Strict linter rules
      - Strict static analyzer rules
      - Reach at least 80% code coverage with unit tests
    - Every PR must pass an integration test suite that contains 13k+ recipe files
      - The test automation scripts will fail if certain thresholds are not met, ensuring we don't regress
  - Data!
    - Automation generates comprehensive statistics
    - Statistics and error logging help identify the current biggest gaps in compatibility

Testing Automation Pipeline



Demo

GitHub Actions CI Results

Questions?