Hashdist Documentation

Release 0.1

Hashdist team

April 11, 2014

CHAPTER 1

User's guide to Hashdist v. 0.2

1.1 Installing and making the *hit* tool available

Hashdist requires Python 2.7 and git.

To start using Hashdist, clone the repo that contains the core tool, and put the `bin`-directory in your `PATH`:

```
$ git clone https://github.com/hashdist/hashdist.git
$ cd hashdist
$ export PATH=$PWD/bin:$PATH
```

The `hit` tool should now be available. You should now run the following command to create the directory `~/.hashdist`:

```
$ hit init-home
```

By default all built software and downloaded sources will be stored beneath `~/.hashdist`. To change this, edit `~/.hashdist/config.yaml`.

1.2 Setting up your software profile

Using Hashdist is based on the following steps:

1. First, describe the software profile you want to build in a configuration file ("I want Python, NumPy, SciPy").
2. Use a dedicated git repository to manage that configuration file
3. For every git commit, Hashdist will be able to build the specified profile, and *cache* the results, so that you can jump around in the history of your software profile.

Start with cloning a basic user profile template:

```
git clone https://github.com/hashdist/profile-template.git /path/to/myprofile
```

The contents of the repo is a single file `default.yaml` which a) selects a *base profile* to extend, and b) lists which packages to include. It is also possible to override build parameters from this file, or link to extra package descriptions within the repository (docs not written yet). The idea is to modify this repository to make changes to the software profile that only applies to you. You are encouraged to submit pull requests against the base profile for changes that may be useful to more users.

To build the stack, simply do:

```
cd /path/to/myprofile
hit build
```

This will take a while, including downloading the source code needed. In the end, a symlink `default` is created which contains the exact software described by `default.yaml`.

Now, try to remove the `jinja2` package from `default.yaml` and do `hit build` again. This time, the build should only take a second, which is the time used to assemble a new profile.

Then, add the `jinja2` package again and run `hit build`. This exact software profile was already built, and so the operation is very fast.

When coupled with managing the profile specification with git, this becomes very powerful, as you can use git to navigate the history of or branches of your software profile repository, and then instantly switch to pre-built versions. [TODO: `hit commit`, `hit checkout` commands.]

If you want to have, e.g., release and debug profiles, you can create `release.yaml` and `debug.yaml`, and use the `-p` flag to `hit` to select another profile than `default.yaml` to build.

1.3 Garbage collection

Hashdist does not have the concepts of "upgrade" or "uninstall", but simply keeps everything it has downloaded or built around forever. To free up disk space, you may invoke the garbage collector to remove unused builds.

Currently the garbage collection strategy is very simple: When you invoke garbage collection manually, Hashdist removes anything that isn't currently in use. To figure out what that means, you may invoke `hit gc --list`; continueing on the example from above, we would find:

```
$ hit gc --list
List of GC roots:
/path/to/myprofile/default
```

This indicates that if you run a plain `hit gc`, software accessible through `/path/to/myprofile/default` will be kept, but all other builds will be removed from the Hashdist store. To try it, you may comment out the `zlib` line from `default.yaml`, then run `hit build`, and then `hit gc` – the zlib software is removed at the last step.

If you want to manipulate profile symlinks, you should use the `hit cp`, `hit mv`, and `hit rm` commands, so that Hashdist can correctly track the profile links. This is useful to keep multiple profiles around. E.g., if you first execute:

```
hit cp default old_profile
```

and then modify `default.yaml`, and then run `hit build`, then after the build `default` and `old_profile` will point to different revisions of the software stacks, both usable at the same time. Garbage collection will keep software for either around.

The database of GC roots is kept (by default) in `~/.hashdist/gcroots`. You are free to put your own symlinks there (you may give them an arbitrary name, as long as they do not contain an underscore in front), or manually remove symlinks.

> **Warning:** As a corollary to the description above, if you do a plain `mv` of a symlink to a profile, and then execute `hit gc`, then the software profile may be deleted by Hashdist.

1.4 Debug features

A couple of commands allow you to see what happens when building.

- Show the script used to build Jinja2:

  ```
  hit show script jinja2
  ```

- Show the "build spec" (low-level magic):

  ```
  hit show buildspec jinja2
  ```

- Get a copy of the build directory that would be used:

  ```
  hit bdir jinja2 bld
  ```

This extracts Jinja2 sources to `bld`, puts a Bash build-script in `bld/_hashdist/build.sh`. However, if you go ahead and try to run it the environment will not be the same as when Hashdist builds, so this is a bit limited so far. [TODO: `hit debug` which also sets the right environment and sets the `$ARTIFACT` directory.]

1.5 Developing the base profile

If you want to develop the `hashstack2` repository yourself, using a dedicated local-machine profile repo becomes tedious. Instead, copy the `default.example.yaml` to `default.yaml`. Then simply run `hit build` directly in the base profile (in which case the personal profile is not needed at all).

`default.yaml` is present in `.gitignore` and changes should not be checked in; you freely change it to experiment with whatever package you are adding. Note the orthogonality between the two repositories: The base profile repo has commits like "Added build commands for NumPy 1.7.2 to share to the world". The personal profile repo has commits like "Installed the NumPy package on my computer".

1.6 Further details

Specifying a Hashdist software profile

CHAPTER 2

Specifying a Hashdist software profile

There are specification file types in Hashdist. The *profile spec* describes *what* to build; what packages should be included in the profile and the options for each package. A *package spec* contains the *how* part: A (possibly parametrized) description for building a single package.

The basic language of the specification files is YAML, see http://yaml.org. Style guide: For YAML files within the Hashdist project, we use 2 space indents, and no indent before vertically-formatted lists (as seen below).

2.1 Profile specification

The profile spec is what the user points the *hit* tool to to build a profile. By following references in it, Hashdist should be able to find all the information needed (including the package specification files). An example end-user profile might look like this:

```
extends:
- name: hashstack
  urls: ['https://github.com/hashdist/hashstack2.git']
  key: 'git:5042aeaaee9841575e56ad9f673ef1585c2f5a46'
  file: debian.yaml

- file: common_settings.yaml

parameters:
  debug: false

packages:
  zlib:
  szlib:
  nose:
  python:
    host: true
  mpi:
    use: openmpi
  numpy:
    skip: true

package_dirs:
- pkgs
- base

hook_import_dirs:
- base
```

extends:

Profiles that this profile should extend from. Essentially this profile is merged on a parameter-by-parameter and package-by-package basis. If anything conflicts there is an error. E.g., if two base profiles sets the same parameter, the parameter must be specified in the descendant profile, otherwise it is an error.

There are two ways of importing profiles:

- **Local**: Only provide the **file** key, which can be an absolute path, or relative to the directory of the profile spec file.
- **Remote**: If **urls** (currently this must be a list of length

one) and **key** are given, the specified sources (usually a git commit) will be downloaded, and the given **file** is relative to the root of the repo. In this case, providing a **name** for the repository is mandatory; the name is used to refer to the repository in error messages etc., and must be unique for the repository across all imported profile files.

parameters:

Global parameters set for all packages. Any parameters specified in the **packages** section will override these on a per-package basis.

Parameters are typed as is usual for YAML documents; variables will take the according Python types in expressions/hooks. E.g., `false` shows up as *False* in expressions, while `'false'` is a string (evaluating to *True* in a boolean context).

packages:

The packages to build. Each package is given as a key in a dict, with a sub-dict containing package-specific parameters. This is potentially empty, which means "build this package with default parameters". If a package is not present in this section (and is not a dependency of other packages) it will not be built. The **use** parameter makes use of a different package name for the package given, e.g., above, package specs for `openmpi` will be searched and built to satisfy the `mpi` package. The **skip** parameter says that a package should *not* be built (which is useful in the case that the package was included in an ancestor profile).

package_dirs:

Directories to search for package specification files (and hooks, see section on Python hook files below). These acts in an "overlay" manner. In the example above, if one e.g., if searching for `python_package.yaml` then first the `pkgs` sub-directory relative to the profile file will be consulted, then `base`, and finally any directories listed in **package_dirs** in the base profiles extended in **extends**.

This way, one profile can override/replace the package specifications of another profile by listing a directory here.

The common case is that base profiles set **package_dirs**, but that overriding user profiles do not have it set.

hook_import_dirs:

Entries for `sys.path` in Python hook files. Relative to the location of the profile file.

2.2 Package specifications

Below we assume that the directory `pkgs` is a directory listed in **package_dirs** in the profile spec. We can then use:

- Single-file spec: `pkgs/mypkg.yaml`

- Multi-file spec: `pkgs/mypkg/mypkg.yaml`, `pkgs/mypkg/somepatch.diff`, `pkgs/mypkg/mypkg-linux.yaml`

In the latter case, all files matching `mypkg/mypkg.yaml` and `mypkg/mypkg-*.yaml` are loaded, and the **when** clause evaluated for each file. An error is given if more than one file matches the given parameters. One of the files may lack the **when** clause (conventionally, the one without a dash and a suffix), which corresponds to a default fallback file.

Also, Hashdist searches in the package directories for `mypkg.py`, which specifies a Python module with hook functions that can further influence the build. Documentation for the Python hook system is TBD, and the API tentative. Examples in `base/autotools.py` in the Hashstack repo.

Examples of package specs are in https://github.com/hashdist/hashstack2, and we will not repeat them here, but simply list documentation on each clause.

In strings; `{{param_name}}` will usually expand to the parameter in question while assembling the specification needed, and are expanded before artifact hashes are computed. Expansions of the form `${FOO}` are expanded at build-time (by the Hashdist build system or the shell, depending on context), and the variable name is what is hashed.

when:

> Conditions for using this package spec, see rules above. It is a Python expression, evaluated in a context where all parameters are available as variables

extends:

> A list of package names. The package specs for these *base packages* will be loaded and their contents included, as documented below.

sources:

> Sources to download. For now, this should be a list with a single item, as implementing a **target** attribute is TBD.

dependencies:

> Lists of names for packages needed during build (**build** sub-clause) or in the same profile (**run** sub-clause). Dependencies from base packages are automatically included in these lists, e.g., if `python_package` is listed in **extends**, then `python_package.yaml` may take care of requiring a build dependency on Python.

build_stages:

> Stages for the build. See Stage system section below for general comments. The build stages are ordered and then executed to produce a Bash script to run to do the build; the **handler** attribute (which defaults to the value of the **name** attribute) determines the format of the rest of the stage.

when_build_dependency:

> Environment variable changes to be done when this package is a build dependency for *another* package. As a special case variable `${ARTIFACT}`

profile_links:

> A small DSL for setting up links when building the profile. What links should be created when assembling a profile. (In general this is dark magic and subject to change until documented further, but usually only required in base packages.)

2.3 Conditionals

The top-level **when** in each package spec has already been mentioned. In addition, there are two forms of local conditionals withi a file. The first one can be used within a list-of-dicts, e.g., in **build_stages** and similar sections:

```
- when: platform == 'linux'
  name: configure
  extra: [--with-foo]

- when: platform == 'windows'
  name: configure
  extra: [--with-bar]
```

The second form takes the form of a more traditional if-test:

```
- name: configure
  when platform == 'linux':
      extra: [--with-foo]
  when platform == 'windows':
      extra: [--with-bar]
  when platform not in ('linux', 'windows'):
      extra: [--with-baz]
```

The syntax for conditional list-items is a bit awkward, but available if necesarry:

```
dependencies:
  build:
    - numpy
    - when platform == 'linux':  # ! note the dash in front
      - openblas
    - python
```

This will turn into either `[numpy, python]` or `[numpy, openblas, python]`. The "extra" – is needed to maintain positioning within the YAML file.

2.4 Stage system

The **build_stages**, **when_build_dependency** and **profile_links** clauses all follow the same format: A list of "stages" that are partially ordered (using **name**, **before**, and **after** attributes). Thus one can inherit a set of stages from the base packages, and only override the stages one needs.

There's a special **mode** attribute which determines how the override happens. E.g.,:

```
- name: configure
  mode: override  # !!
  foo: bar
```

will pass an extra `foo: bar` attribute to the configure handler, in addition to the attributes that were already there in the base package. This is the default behaviour. On the other hand,:

```
- name: configure
  mode: replace  # !!
  handler: bash
  bash: |
      ./configure --prefix=${ARTIFACT}
```

entirely replaces the configure stage of the base package. Finally,:

```
- name: configure
  mode: remove  # !!
```

removes the stage.