teapot Documentation

Release 1.2

Julien Kauffmann

November 28, 2013

Contents

This documentation is about **teapot**, a multi-platform tool to ease fetching, organization and building of third-party softwares.

CHAPTER 1

What is teapot ?

teapot is a Python package that comes with *teapot*, a command-line interface tool. *teapot* reads a YAML file (called the *party file*) which defines the source, the properties, the environment and the build steps for all the third-party libraries to build.

The idea is to add a simple `party.yaml` file inside your project source tree that will describe which third-party libraries it depends on and how to build them.

The first chapter, *The party file*, describes the format of the *party file* and enumerates all the possible options.

The second chapter, *Inside the party*, explains the internals of the `teapot` module, which will allow you to easily write custom filters, extensions, fetchers, unarchivers and to change the complete behavior of **teapot** to perfectly suit your needs.

CHAPTER 2

Why should I use teapot ?

Because you probably have more interesting things to do than dealing with third-party softwares.

Most of the time, people and companies end up writing their own set of scripts to build their dependencies. It can go from a simple *wget* call that fetches precompiled binaries from some server, to more complex systems that download and build them from source and try to do so as reliably as they can.

Writing a script that downloads a *.tar.gz* file, uncompresses it and builds it is really not difficult. But what if you want to handle dependencies between your third party libraries, or desire to support variant builds ? How do you deal with multiple platforms ? How can you react to changes and automatically rebuild what's necessary ? With **teapot**, you just have to write a simple *party file* once and call the *teapot* command once in a while. You can even integrate it into your usual build system since it automatically deals with dependencies and avoids unecessary rebuilds.

How simpler can it get ?

So, will *teapot* build my third-party software for me ?

Yes it will, but you will still have to tell him how exactly.

There are just too many different ways of building software for this to be done without human guidance.

However, *teapot* will make this as painless as it can get by automating all the other things that can be automated.

CHAPTER 4

Why this name ?

No good reason really. I just don't like spending too much time finding catchy names and a *teapot* is a nice tool so... why not ? :)

CHAPTER 5

What's next ?

Here are the chapters you should read if you want to get familiar with *teapot*:

5.1 The *party file*

The *party file* is at the heart of **teapot**. It describes the different third-party softwares to build, and how to build them.

5.1.1 Structure

The *party file* is a YAML file whose root element is a dictionary. While YAML files can make use of a lot of complex data structures, **teapot** only makes use of the common ones, namely:

- dictionaries
- lists
- strings
- booleans
- null

Note: The fact that **teapot** doesn't make use of other data structures doesn't mean you can't use those; you can actually do and use whatever you want when writing custom extensions.

The definition order of all elements in dictionaries is unspecified. This means **teapot** will not care at all in which order you write the keys of a dictionary.

Strings can be any unicode string, however it is **strongly** recommended that you stick with ANSI characters, especially when it comes to indexes.

Attendees

The *attendees* are a first-level element of the root dictionary. They are declared within a dictionary named *attendees* whose each key is the index of an *attendee*, and whose values are the *attendees* themselves.

Here is an example that declares two *attendees*:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
  libcurl:
    source: http://curl.haxx.se/download/curl-7.32.0.tar.gz
```

This example, while perfectly valid, is not quite complete: as they are written, those *attendees* would be able to download and unpack the specified archives, but they don't know how to build the software they constitute.

Here is a more complete *party file* with an *attendee* that actually does something:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
    builders:
      default:
        commands:
          - ./configure --prefix={{prefix}}
          - make
          - make install
```

This *party file* defines completely the way to build *libicon, version 1.14*. The archive will be downloaded from the specified URL, it will be extracted and built with the usuall autotools scenario (*./configure && make && make install*).

In the `./configure` command, you may notice the specific `--prefix={{prefix}}` syntax. This makes uses of an *extension* that will be replaced on runtime by the *prefix* path for this build.

You may find more information on *builders* in the *Builders* section.

An attendee can have the following attributes:

source The source of the attendee. More on that in *Sources*.

filters A list of *filters* that the current execution environment must match in order for the attendee to be active. For instance, one can use filters to specify different attendees for Windows and Linux, within the same *party file*.

builders A dictionary of *builders* that specify what to do with the source code. More on that in *Builders*.

depends A list of names of other *attendees* that this *attendee* depends on for building.

depends can also be a single string in case the *attendee* only depends on one other *attendee*.

prefix The *attendee* specific prefix.

The content of this value is used by the *prefix* extension at runtime.

If *prefix* is a relative path, it will be appended to the *party file*'s prefix.

If *prefix* is an absolute path, it will be taken as it is.

If *prefix* is *True*, it will take the name of the *attendee* as a value. Use this to differentiate builds outputs directories for different *attendees*.

Warning: If the dependency graph is cyclic, *teapot* will notice it before even starting the build and will warn you about the problem.

Sources

The *source* directive in an *attendee* can take several forms.

The simpler form is a *location string*. The possible formats for this depends on the registered *fetchers*.

Here are the default fetchers and their supported formats:

http Fetches an archive from a web URL in a fashion similar to the **wget** command. This is the most commonly used fetcher.

Example formats:

- `http://host/path/archive.zip`
- `https://host/path/archive.zip`

file Fetches an archive from a filesystem path. The path can be either local or a network mount point.

Example formats:

- `~/archives/archive.tar.gz`
- `C:\archives\archive.zip`

github Generates and fetches an archive from a Github-hosted project.

Example formats:

- `github:user/repository/ref`

source can also be a dict of attributes, like so:

```
attendees:
  libiconv:
    source:
      location: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
      type: application/x-gzip
      fetcher: http
      fetcher_options:
      filters: unix
```

All these attributes, except *location* are optional.

location A *location string* as they were just described.

type The mimetype of the archive. Can also be a list of two elements *[mimetype, encoding]* for more complex mimetypes.

fetcher The fetcher to use. Specifying a fetcher disables the automatic fetcher type selection. Specifying a fetcher only makes sense if the location string is ambiguous, which cannot happen with the built-in fetchers.

fetcher_options A dictionary of options for the fetcher. Built-in fetchers do not take any option.

filters A list of filters that the current execution environment must match in order for the source to be active. For instance, one can use filters to specify different sources for Windows and Linux, within the same *attendee*.

For more complex situations, *source* can also be a list of either *location strings* or attributes dictionary (optionaly mixed), like so:

```
attendees:
  libiconv:
    source:
      -
        location: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14_some-variant.tar.gz
        type: application/x-gzip
        fetcher: http
        fetcher_options:
        filters: windows
      - http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
```

Sources are tried in the declaration order for a given *attendee*. In this example, when *teapot* tries to download the archive for the *attendee*, it will first try the first one, only on Windows. If the first one fails (say because of a network error), or if *teapot* is run on a Unix variant, it will skip to the second source.

You may also extend teapot and implement your own fetchers, should you have specific needs.

Unpackers

At some point before the build, *teapot* must convert a downloaded (often compressed) archive into a source tree. This is what *unpackers* are for.

The unpacker selection is done automatically, depending on the mimetype of the downloaded archive. That is, the only way to choose which unpacker to use, is to change the mimetype of the *attendee*.

By default, *teapot* provides the following unpackers:

Tarball unpacker An unpacker that can uncompress tarballs (*.tar.gz* and *.tar.bz2* files).

It recognizes the following mimetypes:

- *application/x-gzip*
- *application/x-bzip2*

Zipfile unpacker An unpacker that can uncompress zip archives (*.zip* files).

It recognizes only the *application/zip* mimetype.

You may also extend teapot and implement your own unpackers, should you have specific needs.

Builders

One of the most important thing to declare into an *attendee*, is its *builders*. A *builder* is responsible for taking an unarchived source tree and creating something by issuing a series of commands.

Builders are declared like so:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
    builders:
      mybuild:
        commands:
          - ./configure --prefix={{prefix}}
          - make
          - make install
```

In this simple example, *teapot* will go into the source tree unpacked from *libiconv-1.14.tar.gz* and will issue the following comma

- `./configure --prefix={{prefix}}`
- `make`
- `make install`

If all of these commands succeed, the build is considered successful as well.

Note: Here `{{prefix}}` is an extension that resolves at runtime as the current prefix for the *builder*. You can learn more about extensions in the *Extensions* section.

One *attendee* can have as many different *builders* as you want it to have. All the *builders* are entries of the *builders* dictionary where the key is the *builder* name, and the value if a dictionary of attributes for the *builder*.

Here is an example of a more complex *attendee*:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
    builders:
      default_x86:
        filters:
          - windows
          - mingw
        environment: mingw_x86
        tags: x86
        commands:
          - ./configure --prefix={{prefix(unix)}}
          - make
          - make install
        prefix: True
        clean_commands:
          - rm -rf {{prefix(unix)}}

      default_x64:
        filters:
          - windows
          - mingw
        environment: mingw_x64
        tags: x64
        commands:
          - ./configure --prefix={{prefix(unix)}}
          - make
          - make install
        prefix: True
        clean_commands:
          - rm -rf {{prefix(unix)}}
```

In this example, we define two builders (*default_x86* and *default_x64*) that have exactly the same build commands.

Both are to be executed if, and only if, MinGW is available in the execution environment. They each make use of a customized *environment* (more on that in *Environments*).

Also note that a tag has been added for every one of them, so that the user can easily choose between x86 and x64 builds when using *teapot*.

Inside the *party file*, the *builder* dictionary supports the following attributes:

commands Can be either a string with a single command to execute or a list of commands to execute.

Commands can contain *extensions* and environment variables that will be substituted upon execution.

clean_commands The list of commands to call when cleaning is requested. *clean_commands* obeys the same rules as command (extensions are replaced as well) however, unlike the regular *commands*, they are executed within the root directory (where the *party file* is located).

environment The environment in which the build must take place.

If no environment is specified, the *default* environment is taken, which is the one the *teapot* command is running in.

You can learn more about environments in the *Environments* section.

tags A list of tags for the *builder*.

Tags can be used later on by the *teapot* command to restrict the *builders* to run dynamically.

One common use for tags is to differentiate *builders* for different build architectures (*x86* and *x64* for instance).

filters A list of filters that the current execution environment must match in order for the *builder* to be active. For instance, one can use filters to specify different builders for Windows and Linux, within the same *attendee*.

prefix The *builder* specific prefix.

The content of this value is used by the *prefix* extension at runtime.

If *prefix* is a relative path, it will be appended to the *attendee*'s prefix.

If *prefix* is an absolute path, it will be taken as it is.

If *prefix* is *True*, it will take the name of the *builder* as a value. Use this to differentiate builds outputs easily for a given *attendee*.

Environments

Environments define the execution environment of a *builder*.

They can be defined either at the attendee level (within a *builder* declaration), or inside the global *environments* dictionary, at the root of *party file*.

An *environment* can inherit from another **named** *environment*.

Here is an example of *party file* that defines environments:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
    builders:
      default_x86:
        environment: mingw_x86
        tags: x86
        commands:
          - ./configure --prefix={{prefix(unix)}}
          - make
          - make install
        prefix: True

      default_x64:
        environment: mingw_x64
        tags: x64
        commands:
          - ./configure --prefix={{prefix(unix)}}
          - make
          - make install
        prefix: True

environments:
  mingw_x86:
    shell: ["C:\\MinGW\\msys\\1.0\\bin\\bash.exe", "-c"]
    inherit: default
    variables:
      PATH: "C:\\MinGW32\\bin:%PATH%"

  mingw_x64:
```

```
shell: ["C:\\MinGW\\msys\\1.0\\bin\\bash.exe", "-c"]
inherit: default
variables:
  PATH: "C:\\MinGW64\\bin:%PATH%"
```

In this example, we define two environments that use the same *shell* (here, *bash* for Windows). They both inherit from the *default* environment and each (re)define the `PATH` environment variable.

An *environment* dictionary understands the following attributes:

shell The *shell* to use.

shell can be a list of command arguments (with the executable as the first argument). This is the recommended way of specifying the *shell* as it is unambiguous.

If *shell* is a string, it will be parsed and split into a list using `shlex.split()`. This method of defining the shell and its arguments can be ambiguous and is therefore **not recommended**.

shell can also be `True` (the default), in which case its value will be taken from the inherited *environment*, if it has one.

If no *shell* is specified, the default one from the system will be taken as specified in `subprocess.call()`.

variables A dictionary of environment variables to set, remove or override.

Each variable can be set to either a string, or to `null` (the YAML equivalent of `None`).

The behavior a null value depends on the value of *inherit*.

If the *environment* inherits its attributes from another *environment*, a null value indicates that the environment variable should be **removed** from the environment. This is **not** equivalent to setting its value to an empty string (in this case the variable would still be part of the environment, but would just be empty).

If the *environment* does not inherit its attributes from another *environment*, a null value indicates that the value for this environment variable should be the one of the execution environment (the environment into which *teapot* was called). If the environment variable was not set within the execution environment, it won't be set in the new environment if its value was `null`.

inherit *inherit* can be null (the default), or it can be the name of a named *environment* to inherit from.

If *inherit* is null, none of the existing environment variables are inherited and only the ones defined in the *variables* attribute will be set.

Note: By default, *teapot* exposes the execution environment through the name `default`.

This `default` environment has all the environment variables that were set right before the call to *teapot* and uses the default system *shell*.

Filters

Filters are a way to differentiate *teapot* execution accross platforms and environments. A *filter* is basically a test whose result is boolean. It answers a simple question like: am on Windows ? Is MinGW available ?

teapot comes with several built-in filters:

Filter	Role
windows	Check that *teapot* is currently running on Windows.
linux	Check that *teapot* is currently running on Linux.
darwin	Check that *teapot* is currently running on Darwin (Mac OS X).
unix	Check that *teapot* is currently running on UNIX (Linux or Darwin).
msvc	Check that Microsoft Visual Studio is actually available in the current environment. It usually means *teapot* was started from a MSVC command shell.
msvc-x86	Check that Microsoft Visual Studio x86 is actually available in the current environment. It usually means *teapot* was started from a MSVC x86 command shell.
msvc-x64	Check that Microsoft Visual Studio x64 is actually available in the current environment. It usually means *teapot* was started from a MSVC x64 command shell.
mingw	Check that MinGW is available in the current environment. The filter will try to find *gcc.exe*.

Note: When defining several *filters* in an *attendee*, a *source* or a *builder*, note that **all** filters must be verified for the validation to pass.

You may also define your own filters, see *Writing extension modules*.

Extensions

Extensions are simple functions, that optionally have parameters, which can occur in a *builder* command.

For instance the *prefix* extension is resolved at runtime and replaced with the complete prefix (as defined at the root of the *party file*, the *attendee* and the *builder*).

Here is an example:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz
    builders:
      default_x86:
        filters: mingw
        commands:
          - ./configure --prefix={{prefix(unix)}}
          - make
          - make install
        prefix: True
```

In this example, designed to run from within a MSys environment on Windows, we make use of the *prefix* extension and we supply the *style* parameter. Upon runtime, the expression gets replaced with the UNIX-style path to the prefix, as defined in the *party file*.

Valid syntaxes for calling extensions within commands are:

```
{{extension}}              # No parameters.
{{extension()}}            # No parameters. No difference with the first call.
{{extension(arg1)}}        # Call with one parameter.
{{extension(arg1,arg2)}}   # Call with two parameters.
{{extension(,arg2)}}       # Call with two parameters, the first one being omitted.
{{extension(arg1,,arg3)}} # Call with three parameters, the second one being omitted.
```

teapot comes with several built-in extensions:

Extension	Parameters	Role
root	style	Get the absolute path to the root of the *party file*. Returns the complete path, in an operating system specific manner. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments.
prefix	style	Get the complete prefix for the current attendee/builder. Returns the complete path, in an operating system specific manner. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments. *prefix* can contain extensions, as long as it doesn't call itself directly, or indirectly.
prefix_for	attendee, builder, style	Get the complete prefix for the specified attendee/builder. You must at least specify the *attendee* parameter. Returns the complete path, in an operating system specific manner. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments. *prefix_for* can contain extensions, as long as it doesn't call itself directly, or indirectly.
current_attendee		Returns the current attendee name.
current_builder		Returns the current builder name.
current_archive_path	style	Returns the current archive path. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments.
current_source_tree_path	style	Returns the current source tree path. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments. Since source trees are copied to a temporary location before the build, this is **not** the path were the build actually takes place.

You may also define your own extensions, see *Writing extension modules*.

Other settings

teapot runs with the following defaults:

Parameter	Default value	Meaning

cache_path `~/.teapot.cache` (UNIX) The path where the archives are downloaded to.

`%APPDATA%/teapot/cache` (Windows)

build_path `~/.teapot.build` (UNIX) The path where the builds take place.

`%APPDATA%/teapot/build` (Windows)

prefix `install` The default *party file* prefix that gets prepended to all *attendees* prefixes.

These settings are to be set at the root of the *party file*, like so:

```
attendees:
  libiconv:
    source: http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz

cache_path: cache
build_path: build
prefix: install
```

Depending on your project, you may want to set the *cache_path* to a more local location (you may choose to add them to version control for instance).

Writing extension modules

teapot was designed from the start to be extensible.

Using the *extension_modules* attribute at the root of *party file*, you can extend *teapot* any way you want.

Those extensions modules are regular Python modules into which you can define *filters*, *extensions*, *environments* or anything else you want.

The *extension_modules* attribute is a dictionary located at the root of the *party file* where keys are the shortnames for the modules, and the values are the path to those modules:

```
extension_modules:
  myfilter: modules/myfilter.py
  myenvironment: modules/myenvironment.py
```

To get more details about how to write filters, extensions and environments, take a look at *Inside the party*.

5.1.2 Using *teapot*

teapot is the command line tool that ships with *teapot*.

```
$ teapot --help
usage: teapot [-h] [-d] [-v] [-p PARTY_FILE]
              {clean,fetch,unpack,build} ...

Manage third-party software.

positional arguments:
  {clean,fetch,unpack,build}
                        The available commands.
    clean               Clean the party.
    fetch               Fetch all the archives.
    unpack              Unpack all the fetched archives.
    build               Build the archives.

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Enable debug output.
  -v, --verbose         Be more explicit about what happens.
  -p PARTY_FILE, --party-file PARTY_FILE
                        The party-file to read.
```

By default, *teapot* looks for a file named `party.yaml` in the current directory. You may change the location of this file by using the `--party-file` option.

The *clean* command

teapot fetches the sources archives and stores them in the *cache* directory. It also build attendees and stores the temporary results inside the *build* directory.

Use `teapot clean` to clean either the *cache* or the *build* directory (or both).

The use of this command in normally not needed as *teapot* knows how to compute dependencies and detect changes automatically.

```
$ teapot clean --help
usage: teapot clean [-h] {cache,build,all} ...

positional arguments:
  {cache,build,all}  The available commands.
    cache            Clean the party cache.
    build            Clean the party build.
    all              Clean the party cache and build.

optional arguments:
  -h, --help         show this help message and exit
```

The *clean cache* command

Cleans the *teapot* cache directory, where the source archives are stored.

Use this command if, for whatever reason you think the archive cache was corrupted.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean cache --help
usage: teapot clean cache [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.

optional arguments:
  -h, --help  show this help message and exit
```

The *clean build* command

Cleans the *teapot* build directory, where the build results are stored.

Use this command if, for whatever reason you think the build results were corrupted.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean build --help
usage: teapot clean build [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.
```

```
optional arguments:
  -h, --help  show this help message and exit
```

The *clean cache* command

Cleans the *teapot* cache and build directories.

Use this command if, for whatever reason you want to reset the status of your current *teapot* project.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean all --help
usage: teapot clean all [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.

optional arguments:
  -h, --help  show this help message and exit
```

The *fetch* command

Fetches the source archives of the specified *attendees*.

`teapot fetch` makes sure all the source archives are downloaded for the specified attendees.

If no *attendee* is specified, the source archives for all *attendees* are fetched.

By default, this command only fetches archives that weren't already downloaded. Use the `--force` option to force the download of all *attendees*.

```
$ teapot fetch --help
usage: teapot fetch [-h] [-f] [attendee [attendee ...]]

positional arguments:
  attendee     The attendees to fetch.

optional arguments:
  -h, --help   show this help message and exit
  -f, --force  Fetch archives even if they already exist in the cache.
```

The *unpack* command

Unpacks the fetched source archive to prepare for a build.

If no *attendee* is specified, all the attendees are unpacked.

```
$ teapot unpack --help
usage: teapot unpack [-h] [-f] [attendee [attendee ...]]

positional arguments:
  attendee     The attendees to unpack.

optional arguments:
  -h, --help   show this help message and exit
  -f, --force  Unpack archives even if they already exist in the build.
```

This step is usually not required as it performed automatically whenever needed. Use it when you don't want to build right away but want the next build to be as fast as possible.

Calling *unpack* automatically fetches the source archives if they are not present.

The *build* command

Builds the attendees.

If no *attendee* is specified, all the attendees are built. If a list of *attendees*<*attendee*> is specified, only those attendees and the ones they depend on will be built.

```
$ teapot build --help
usage: teapot build [-h] [-t tag] [-u] [-f] [-k] [attendee [attendee ...]]

positional arguments:
  attendee              The attendees to build.

optional arguments:
  -h, --help            show this help message and exit
  -t tag, --tags tag    The tags to build.
  -u, --force-unpack    Delete and reunpack all source tree directories before
                        attempting a build.
  -f, --force-build     Run all builders even if their last run was successful.
  -k, --keep-builds     Keep the build directories for inspection.
```

By default, all variants from all builders are taken. You may specify the `--tags` option to build only specific variants (like *x86* or *x64* for instance).

Only the builders that didn't succeeded the last time or the one that changed since the last build are run. To change that behavior, specify the `--force-build` option.

teapot will not try to re-unpack archives that were already unpacked unless `--force-unpack` is specified.

Temporary build directories are deleted automatically whenever a build terminates (either with a success or a failure), unless the `--keep-builds` option is specified. In that case, the build directory remains until the build gets restarted.

5.2 Inside the party

This chapter describes how to write custom module-extensions for *teapot*.

5.2.1 Filters

We already seen in *Filters* what a *filter* is. Now is the time to write your owns !

A *filter* is a simple function that takes no parameters and returns a boolean value.

To register a new *filter*, use the `teapot.filters.decorators.named_filter()` decorator.

class `teapot.filters.decorators.`**`named_filter`**(*name*, *depends=None*, *override=False*)

Registers a function to be a filter.

`__init__`(*name*, *depends=None*, *override=False*)

Registers the function with the specified name.

If another function was registered with the same name, a *DuplicateFilterError* will be raised, unless *override* is truthy.

***depends* can be either:**

- *None*, if the filter does not depend on any other filter.
- A *str* instance, being the registered name of another filter to depend on.
- A *list* of *str* instances, being the list of registered names of other filters to depend on.

If a filter depends on other filters, those will be checked *before* the actual filter gets run.

Here is an example of some built-in *filters*:

```
import os
import sys

from teapot.filters.decorators import named_filter

@named_filter('windows')
def windows():
    """
    Check if the platform is windows.
    """

    return sys.platform.startswith('win32')

@named_filter('msvc', depends='windows')
def msvc():
    """
    Check if MSVC is available.
    """

    return 'VCINSTALLDIR' in os.environ
```

5.2.2 Extensions

As seen in *Extensions*, an *extension* is a function, that always takes a *builder* argument, optionally takes string parameters and returns a string.

Extensions are to **teapot** what macros are to the C language.

To register a new *extension*, use the `teapot.extensions.decorators.named_extension()` decorator.

Here is an example of some built-in *extensions*:

```
import os
import sys

from teapot.path import windows_to_unix_path
from teapot.extensions.decorators import named_extension

@named_extension('prefix')
def prefix(builder, style='default'):
    """
    Get the builder prefix.
    """

    result = os.path.join(builder.attendee.party.prefix, builder.attendee.prefix, builder.prefix)

    if sys.platform.startswith('win32') and style == 'unix':
        result = windows_to_unix_path(result)
```

```
    return result
```

Note that the function **must** always have first *builder* argument that will be valued with the current `teapot.builders.Builder` instance.

5.2.3 Fetchers

Fetchers are responsible for downloading or copying the source archives from a specified location.

To define a new fetcher, just derive from `teapot.fetchers.base_fetcher.BaseFetcher`.

Here is an example with the built-in *file* fetcher:

```
import os
import shutil
import mimetypes

from teapot.fetchers.base_fetcher import BaseFetcher


class FileFetcher(BaseFetcher):

    """
    Fetchs a file on the local filesystem.
    """

    shortname = 'file'

    def read_source(self, source):
        """
        Checks that the `source` is a local filename.
        """

        if os.path.isfile(source.location):
            self.file_path = os.path.abspath(source.location)

            return True

    def do_fetch(self, target):
        """
        Fetch a filename.
        """

        archive_path = os.path.join(target, os.path.basename(self.file_path))

        archive_type = mimetypes.guess_type(self.file_path)
        size = os.path.getsize(self.file_path)

        self.progress.on_start(target=os.path.basename(archive_path), size=size)

        shutil.copyfile(self.file_path, archive_path)

        # No real interactive progress to show here.
        #
        # This could be fixed though.

        self.progress.on_update(progress=size)
```

```
        self.progress.on_finish()

        return {
            'archive_path': archive_path,
            'archive_type': archive_type,
        }
```

Callbacks

All callback classes derive from `teapot.fetchers.callbacks.BaseFetcherCallback`.

5.2.4 Unpackers

Unpackers are responsible for extracting the content of the source archives into an exploitable source tree.

To define a new unpacker, just derive from `teapot.unpackers.base_unpacker.BaseUnpacker`.

class `teapot.unpackers.base_unpacker.`**`BaseUnpacker`**(*attendee*)

Base class for all unpacker classes.

If you subclass this class, you will have to re-implement the `do_unpack()` method to provide your specific unpacker logic.

If you desire to attach your unpacker to certain mimetypes, please do so by defining the class-level member *types* that must be a list of couples *(mimetype, encoding)*.

`do_unpack`()

Unpack an archive.

The archive to unpack can be reached at *self.archive_path*.

This method must return a dict with the following keys:

- *source_tree_path*: The extracted source tree path.

It must raise an exception on error.

You can provide feedback on the unpacking operation by calling *self.progress.on_start*, *self.progress.on_update* and *self.progress.on_finish* at the appropriate time.

See `teapot.unpackers.callbacks.BaseUnpackerCallback` for further details.

Here is an example with the built-in *tarball* unpacker:

```
from teapot.unpackers.base_unpacker import BaseUnpacker

import os
import tarfile


class TarballUnpacker(BaseUnpacker):

    """
    An unpacker class that deals with .tgz files.
    """

    types = [
        ('application/x-gzip', None),
        ('application/x-bzip2', None),
    ]
```

```
def do_unpack(self):
    """
    Uncompress the archive.

    Return the path of the extracted folder.
    """

    if not tarfile.is_tarfile(self.archive_path):
        raise InvalidTarballError(archive_path=self.archive_path)

    tar = tarfile.open(self.archive_path, 'r')

    # We get the common prefix for all archive members.
    prefix = os.path.commonprefix(tar.getnames())

    # An archive member with the prefix as a name should exist in the archive.
    while True:
        try:
            prefix_member = tar.getmember(prefix)

            if prefix_member.isdir:
                break

        except KeyError:
            pass

        new_prefix = os.path.dirname(prefix)

        if prefix == new_prefix:
            raise TarballHasNoCommonPrefixError(archive_path=self.archive_path)
        else:
            prefix = new_prefix

    source_tree_path = os.path.join(self.attendee.build_path, prefix_member.name)

    self.progress.on_start(count=len(tar.getmembers()))

    for index, member in enumerate(tar.getmembers()):
        if os.path.isabs(member.name):
            raise ValueError('Refusing to extract archive that contains absolute filenames.')

        self.progress.on_update(current_file=member.name, progress=index)
        tar.extract(member, path=self.attendee.build_path)

    self.progress.on_finish()

    return {
        'source_tree_path': source_tree_path,
    }
```

Callbacks

All callback classes derive from `teapot.unpackers.callbacks.BaseUnpackerCallback`.

5.2.5 Party post-actions

post-actions are simple function that takes a `teapot.party.Party` instance as a parameter, and that gets executed right after a such instance was initialized.

You can register a *post-action* using the `teapot.party.Party.register_post_action()` decorator.

Here is an example a built-in *post-action* that registers the defaut environment:

```
@Party.register_post_action
def add_default_environment(party):
    """
    Add the default environment to the party.
    """

    party.environment_register.register_environment(DEFAULT_ENVIRONMENT_NAME, create_default_environ
```

5.3 Glossary

party file The party file is a YAML file, named *party.yaml* that can be located anywhere.

Within the party file, all paths are relative to the party file directory.

attendee An attendee is a fancy name for a third-party software to build.

A *party file* can contain as many attendees as you like, and different attendees can even represent the same third-party software if that makes sense in your situation.

source A source designates the location and the method where and how to fetch the source files for an *attendee*. While the most common case is downloading a file using HTTP, one can also copy a file locally, through a network share or from Github.

fetcher A fetcher is the entity that is responsible for handling a specific type of source.

Usually, fetchers are smart enough to recognize sources from their format and you should not have to care too much about them.

unpacker An unpacker is the entity that is responsible for turning a compressed archive (Zip file or tarball for instance) into a source tree.

builder A builder is the list of commands to execute in order to transform the attendee source into a compiled set of binaries (or whatever a build process can produce).

Builders rely a lot on *environments*.

environment An environment is a set of environment variables, shell value and inheritance parameters that wraps one or several builds.

Environments define the tools to use for a given build, and their options.

filter A filter is an entity whose role is to check if the current execution environment matches a series of criterias.

For instance, the *windows* filter checks that *teapot* has been run on Windows. Another example is the *mingw* filter whose role is to check that MinGW is currently available in the execution environment.

teapot *teapot* is the name of the command-line tool that implements all teapot logic.

shell A shell is a command line interpreter that will execute the different commands of a builder.

extension An extension is an entity the resides in builder commands and that gets replaced when the command is evaluated.

Extension are python function that can optionally take parameters.