JBackup

The Pythonic Templated Backup System

ABSTRACT

JBackup (working title) is a Python-based system for backing up files and directories. A set of *actions* will define what the system can do, and a set of *rules* will define what files said actions act upon.

Synopsis

JBackup is a backup system for developers who want a convenient way to compress their local repositories and copy them elsewhere. It is built on the idea that certain “modules” called *actions* should provide a template for which *rules* can be supplied. (See **Terminology** for an overview of what actions and rules are.)

Terminology

Terms used in this document are defined thusly:

- *Action:* a script which implements a specific function. Actions are, by design, generic so that rules can provide data to specialize them.
- *Property*: in the context of actions, a variable that is available to be modified by rules. For clarity’s sake, this document will also use the full name: action-property.
- *Rule:* a configuration file that provides values to an action. It is written in a format that supports sections (in this case, TOML). Each section corresponds to an action, and its values are read by said action and used as parameters. The primary rule class is a client in the adapter pattern; the underlying config file is the adapter (see **Rules** below).
- *Data path:* the current path to a directory under which actions and rules are placed. Actions and rules are each placed in their own subdirectory of the same name. The data path is selected from a list of two predefined paths, (1) a system-wide path under `/usr/local/etc`, and (2) the user’s `.local/etc` path. If the user has root privileges the system path is used, otherwise the user path is used.

User Interface

JBackup is a command-line application. It supports several subcommands, each with their own set of options.

The user can create new actions with the `create-action` subcommand.

```
   $ jbackup create-action ACTION
```

`ACTION` is the name of the action they want to create. When invoked, this subcommand copies a template into the actions subdirectory under the *data path*.[1] The path to the newly created action is printed out.

The user can create a new rule with a chosen name.

```
   $ jbackup create-rule [-f|--format FORMAT] RULE
```

`RULE` is the name of a rule they want to create. A config file of the selected format (dictated by -f) is written to the rules subdirectory under the current data path. The current default is TOML. The path to the newly created rule is printed.

The user can run an action with one or more rules using this commandline:

```
   $ jbackup do ACTION RULE ...
```

`ACTION` is the action they wish to run. They supply one or more rules to said action. The action is run with each rule supplying it with values for its properties, and any output from the action is printed.

The user can query information from JBackup with this commandline:

```
   $ jbackup show ACTION
```

`ACTION` is, of course, the action they want to query. The output of this command is the documentation of the action: the action class' docstring, as well as any properties and their docstrings (see **Action Properties** below).

The user can locate a specific action or rule with this command:

```
   $ jbackup locate [-r|--rule] ACTION_OR_RULE
```

The argument is a rule unless `-r` is provided, in which case it is a rule. The output is a single line: the absolute path to the file defining the action or rule.

This command is used by the completion script under `data/`:

```
   $ jbackup complete [--firstarg] CWORD ARG ...
```

If `--firstarg` is provided, then a list of subcommands and the options shown below are returned. *CWORD* is the value of `$COMP_CWORD` in bash completion scripts; that is to say, the current index on the commandline. The following *ARG*s are parts of the commandline provided so far.

JBackup has options for listing information about JBackup:

```
   $ jbackup --list-actions
   $ jbackup --list-rules
   $ jbackup --path
   $ jbackup --levels
```

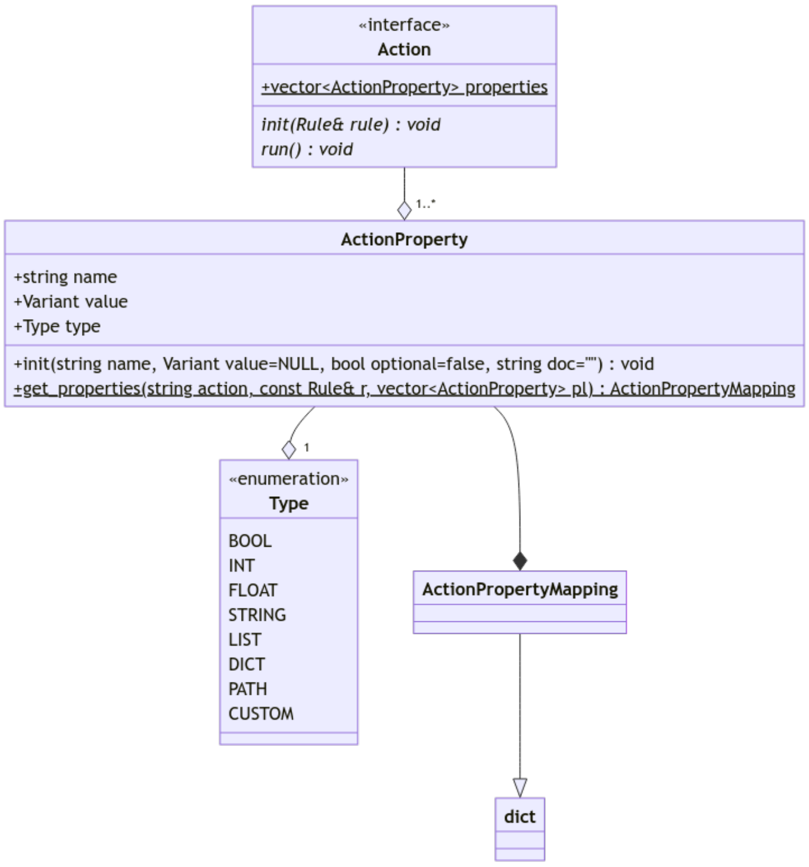
1 Defined on page 1.

The first two commands list currently available actions and rules, respectively. They are separated into categories of system and user-level actions/rules. The third command prints the data path. The fourth command prints the available logging levels. All these commands exit after doing their task.

Technical Specification

Actions

An action is a generic interface for doing a certain job; it is implemented as a script containing a class. The class defines the behavior and properties of the action. It holds an array of *action-properties*, which can then be set by a rule.



«interface»
Action
+vector<ActionProperty> properties
init(Rule& rule) : void
run() : void
1..*
ActionProperty
+string name
+Variant value
+Type type
+init(string name, Variant value=NULL, bool optional=false, string doc="") : void
+get_properties(string action, const Rule& r, vector<ActionProperty> pl) : ActionPropertyMapping
1
«enumeration»
Type
BOOL
INT
FLOAT
STRING
LIST
DICT
PATH
CUSTOM
ActionPropertyMapping
dict

Figure 1: Action Protocol

Actions are based around the protocol design pattern. `Action` is the protocol defining the structure of an action class. Each action has a list of properties. It has a `run()` method, which is responsible for enacting the action's behavior.

The constructor accepts a `Rule` instance; from this rule, `Action` retrieves values to assign to its properties.

The static method `get_properties()` is used by the action class to get a mapping of action-properties. (See `_template.py` for to see this in action.)

Actions are loaded with the `load_action()` function. Internally, it uses the `loader` submodule to load the action file as a module, then it extracts the class from the module. Refer to the section titled **Loader** for more information.

When the action class is extracted, it's used to instantiate an object, and that object is provided a rule object to set its property values. (See Figure 1.)

Action Properties

Action properties are represented by the `ActionProperty` class. They each have a name, a value, and a type identifier. Besides those, each action-property has a documentation string. Its documentation string, name and property type, can be accessed as instance attributes.

> `ActionProperty` has an internal `__str__()` method which returns a string containing the type, name, and documentation of the property. It is used by the `show` subcommand.

An action contains one or more properties collected into a list. For each property, unless it is set to be optional (via the constructor), then an error is raised if a rule does not set its value.

`ActionProperty` has an option to specify a list of property types; if it is set, type validation is turned on. When the value is set, if its type (`PropertyType`) does not match one of the types in the aforementioned list, `PropertyTypeError` is raised.

Loader

The `loader` module is used to dynamically load scripts in as modules. The module consists of the `ModuleProxy` class and a single function, `load_module_from_file()`.

`load_module_from_file()` expects a string or a path-like object, a path to a Python script which can be loaded like a module, and a string providing a name for the module.

The function returns a `ModuleProxy` object holding the newly-loaded module. `ModuleProxy` mocks having the module's attributes via the dot operator.

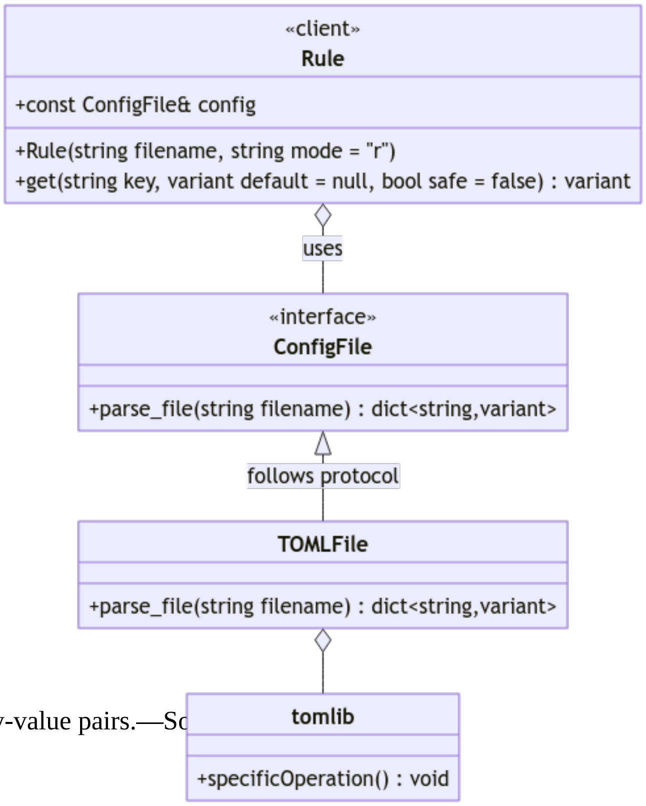
A flag can be set, and it's called the "safe" flag. If the flag is set, then `AttributeError` is not raised where it normally would be, which is to say, when a nonexistent attribute is referenced.

Rules

A rule is a configuration file which contains values pertaining to actions. Currently rules are written in *TOML*.[2] Each section is named after the action in question, and its values correspond to the properties of said action.

Rules are represented by the `Rule` class. It accepts two arguments, a string path pointing to a file, and an optional string mode, either `r` or `w`. If the mode is `w`, an example rule file is written at the designated path.

Rules are implemented using the adapter pattern. `Rule` is the client. `ConfigFile` is the interface


«client»
Rule
+const ConfigFile& config
+Rule(string filename, string mode = "r")
+get(string key, variant default = null, bool safe = false) : variant
uses
«interface»
ConfigFile
+parse_file(string filename) : dict<string,variant>
follows protocol
TOMLFile
+parse_file(string filename) : dict<string,variant>
tomlib
+specificOperation() : void

2 TOML files are composed of sections. Each section has key-value pairs.—Sc

which defines functions for parsing files and getting values. `TOMLFile` is an adapter to the `tomllib` library, which does the actual parsing.[3]

Templates

The template subpackage handles the creation of actions and rules; actions via a template, and rules via a simple file-write operation. Actions and rules are written to their respective paths with appropriate file names.

Actions are created via `write_action_file()`. It accepts a string or a path-like object designating a path to the file to be created. Its second argument is a string, the name of the action being created. This is used in substituting text inside the template. The path to the newly created file is returned as a string.

Rules are created via `write_rule_file()`. It accepts the same arguments as `write_action_file`, except they are applied to the newly created rule file. Said file's path is returned as a string.

Logging

The `logging` module contains functions and classes that create and manipulate loggers. Currently it is built off of Python's inherit `logging` module.

To get a logger, call the function `get_logger()`. It accepts three arguments, the first being a string name for the logger. The other two are optional.

> **Side note**: `logging` has a hierarchy for its loggers, with its root being the one with the name "root" or even an empty string. In JBackup, however, the logger named "jbackup" is considered the root logger, for all intents and purposes. The package initializer calls an internal function in this module to initialize the root logger.
>
> Every logger created with this function is a descendant of "jbackup". If the name contains dots, then each successive name is the direct descendant of the one before it. For example, "foo.bar" indicates that "bar" is a direct descendant of "foo".

The `_StackHandler` class is a custom handler for loggers which is only available in debug mode. To use it, call `add_handler()` with the *kind* "stack". The debug-only function `get_record_tuple()` returns a tuple with the logger name, the logging level, and the message. It's used in the unit tests for the `logging` module, so see that for examples.

For more information, please see the documentation using `pydoc`.

3 `tomllib` is unavailable below Python version 3.11. In such cases, it becomes an alias for `tomli.`