Mt Wilson

Setup

# Background

The setup facility for an extensible application must enable plugin-writers to define their own setup tasks for plugin features. It must support a rich user experience by making it possible for the user to know whether a setup task succeeded and also why it failed, and when running multiple setup tasks it must be able to share its progress. Because the application may have many setup tasks, and it’s not known in advance which setup tasks will be expensive in time, CPU, network, disk, or other resources, it must be possible for setup tasks to self-regulate and determine whether their pre-conditions are met before executing, determine whether their post-conditions are met after executing, and automatically skip execution if it’s not necessary. The user must also be able to force execution if required.

## Setup tasks are not commands

Commands are routinely issued with different inputs in order to get work done, for example adding a new user, copying or moving files, or generating reports. When a user runs a command to copy a file, for example, the user expects the file to be copied every time the command is run, unless an option is specified to skip the action under some circumstances.

Setup tasks are expected to run only when needed, typically just one time unless something changes that requires them to run again. Therefore running a setup task multiple times should result in the action happening just once, unless an option is specified to force execution. In other words, given a set of tasks, it should be possible to check each one to see if it needs to be done, or if it already completed successfully and can be skipped. Setup tasks correspond to a list of things that can be “checked off” when done and normally don’t need to be repeated.

# Architecture

The mtwilson-setup project defines a SetupTask interface, a LocalSetupTask abstract implementation with helper methods, and a Setup command which is able to manage the execution of one or more setup tasks. All setup should be performed using implementations of SetupTask in order to provide the user a consistent interface to the required setup capabilities.

Setup tasks can be anything that is required for Mt Wilson or a Mt Wilson feature to function.

Characteristics of setup tasks:

* Their work can be validated to ensure it was successfully completed
* They may access any resource such as local disk, database, or the network
* No user interface for input or display; setup tasks communicate via the API

Setup tasks are always run as part of an application which handles the user interface, such as a console, desktop, or web application - but setup tasks themselves cannot ask the user for input or display messages; they only report on what configuration is missing or what errors they encountered and the application is responsible for interacting with the user.

Setup tasks can access the current configuration and the environment for input, but this should not be abused because it complicates the user experience by requiring the user to have certain variables set before running setup, and therefore requires the use of an “environment file” to keep all those settings as a persistent pre-configuration of the application, or repeat the selections in a user interface. Therefore the use of environment variables should be limited to those settings that are persistent across invocations of the setup manager, such as choice of IP address or port number, or security settings.

Tasks that require a different input each time, for example adding a new user, should be implemented as commands with parameters instead of as setup tasks.

# Setup API

Any feature may have setup tasks that may have to run and complete before it can be activated.

Currently each setup task is responsible for checking that its dependencies are met before running. This can be done by checking for existence of files, folders, database tables, and classes on the classpath (via Class.forName).

A subclass defines configure(), validate(), and execute() methods, and optional JavaBean-style getters and setters for its configuration parameters.

The configure() method gathers available configuration (set via the setters) or generates suitable defaults. If a default cannot be generated for a required parameter, or if there is something unsuitable about parameters provided via the setters, the configure( ) method logs “configuration faults” which can later be displayed to the user. If the configure() method does not log any faults it is considered to be successful. The configure() method should not take any action to “save” the configuration - all the configuration parameters for the setup task should be modeled as getters and setters; the application is responsible for reading these after calling configure() in order to save the resulting configuration somewhere.

The validate() method is called whenever the application wants to find out if the setup task has successfully completed. This might happen immediately after execute(), or it might happen far in the future to validate an existing setup. The validate() method can assume that configure() has already been called. The validate() method generally checks that things are where they are supposed to be, are in the right format, have the right permissions, etc. The validate() method can log “configuration faults” if required configuration is not available for it to check anything, and “validation faults” for any other issues - for example if a file is missing from its configured location, or is the wrong format, etc. These faults may be displayed to the user. The validate() method should **not** take any action to correct any faults.

The execute( ) method is called to perform the setup task itself. The setup task can assume that configure() was called before execute(). Examples of execute() might be to generate a file, apply patches or SQL scripts, and change file permissions.

# Development Roadmap

**Dependencies**. At this time there is no implementation for dependencies between setup tasks. It would be useful if a setup task can provide a list of its setup dependencies so the manager can ensure that they are executed first in order to provide the best chance for this setup task to complete successfully. This might mean auto-execution (by the manager) of dependencies when executing all setup tasks, or it might mean that dependencies are only checked if a setup task reports a configuration or validation fault to attempt to auto-correct the error. The facility would work best when the setup manager is allowed to execute all tasks. The user would still have fine-grained control by invoking the setup manager on a specific task or ordered set of tasks and specifying that dependencies should not be followed.

**Plugins**. It should be possible for the user to run a command which automatically detects all available setup tasks, forms the dependency tree, and starts execution. That way a user can run a general setup request after installing a plugin and not have to read any documentation to find out which setup tasks should be run after installing that plugin. This might translate simply to the setup manager using the extensions facility to detect all available setup tasks and using that as the input. It’s possible to make the dependencies on specific tasks or on artifacts, such as “does this file exist” or “is there at least one item in that list”, or on symbols that can be set by each task’s validate method so that the password vault task can indicate that a password vault now exists (using a symbol) and that other tasks can use the password vault API to add a password, without making those other tasks have to know where to look for the password vault on disk or in a database in order to know that it’s available.

**Configuration**. It should be possible for setup tasks to declare which configuration settings they require. Currently they might use some public static final String constants internally, but there is no uniform way to inform the user what needs to be known before running them, and also it’s possible that one task’s configuration requirement will be provided by another task so it should be possible to mark that as well so user doesn’t bother to look at it. So this can be as easy as adding an interface for listing configuration settings that may be used as input (and nothing else - and this can be enforced by the setup manager because it could actually provide a configuration subset with just those settings)

**Interface**. The Setup interface should be revised as follows: 1) combine configuration faults and validation faults into a single list of faults, and implement the Faults interface (from mtwilson-util-validation) so the faults list is available from a method called getFaults(); 2) implement the Configurable interface (mtwilson-util-configuration) so that configuring a setup task is done by calling configure(Configuration) - at least that provides the settings that the setup task declared that it needs, and it can get anything else on its own; 3) there should be a method to declare applicable configuration settings, maybe List<Property> getProperties() where Property has a name and a default value, or maybe Configuration getDefaultConfiguration() which provides names and default values and the setup manager can just layer it with the actual provided configuration when it calls configure(), or maybe String[] getConfigurationKeys()

**Undo API**. Some things that are done by setup tasks such as creating files, copying files, etc. may be possible to reverse by storing backup copies of target files before they are replaced. Other things its not possible to undo (getting a public key certified - you can delete the certificate or put it on a CRL but you can’t completely undo this because the CA tracks serial numbers, and it’s not going to roll back). So copying, renaming, patching, deleting, these are actions that should implement an Undo interface. It may not be necessary to do this if we use either Linux distribution package manager or Windows MSI generator which already have this feature for the installation portion.