When Documentation

Release 0.9

Francesco Garosi

Jun 13, 2017

Contents

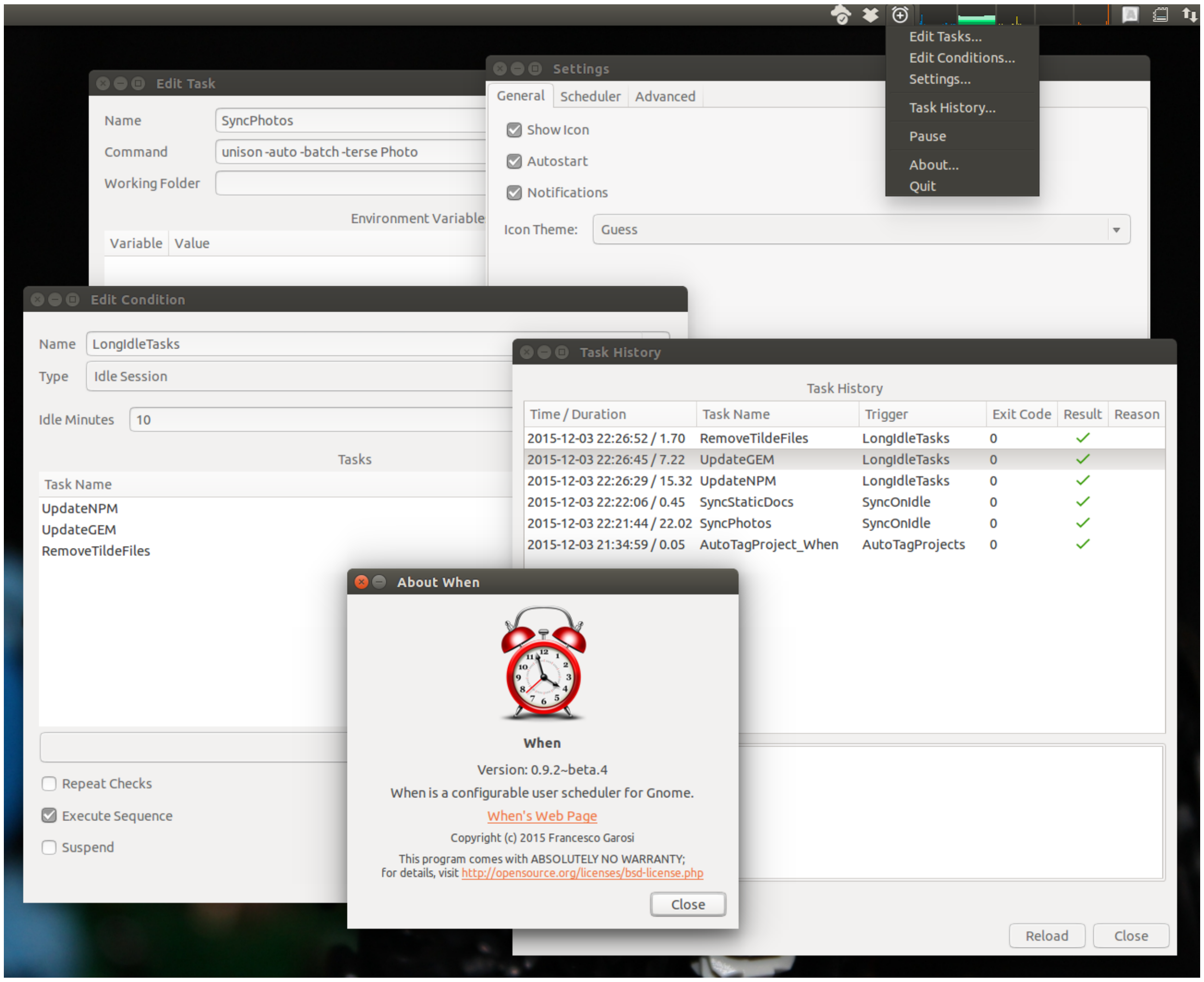
Contents:

CHAPTER 1

Introduction

When is a configurable user task scheduler for modern Gnome environments. It interacts with the user through a GUI, where the user can define tasks and conditions, as well as relationships of causality that bind conditions to tasks. When a condition is bound to a task, it is said to trigger a task.

When is available in source or packaged form on GitHub.



Edit Task

Name: SyncPhotos
Command: unison -auto -batch -terse Photo
Working Folder:
Environment Variable
Variable | Value

Settings

General | Scheduler | Advanced

☑ Show Icon
☑ Autostart
☑ Notifications
Icon Theme: Guess

Edit Condition

Name: LongIdleTasks
Type: Idle Session
Idle Minutes: 10

Tasks

Task Name
UpdateNPM
UpdateGEM
RemoveTildeFiles

☐ Repeat Checks
☑ Execute Sequence
☐ Suspend

Task History

Time / Duration	Task Name	Trigger	Exit Code	Result	Reason
2015-12-03 22:26:52 / 1.70	RemoveTildeFiles	LongIdleTasks	0	✓	
2015-12-03 22:26:45 / 7.22	UpdateGEM	LongIdleTasks	0	✓	
2015-12-03 22:26:29 / 15.32	UpdateNPM	LongIdleTasks	0	✓	
2015-12-03 22:22:06 / 0.45	SyncStaticDocs	SyncOnIdle	0	✓	
2015-12-03 22:21:44 / 22.02	SyncPhotos	SyncOnIdle	0	✓	
2015-12-03 21:34:59 / 0.05	AutoTagProject_When	AutoTagProjects	0	✓	

Reload | Close

About When

When

Version: 0.9.2~beta.4

When is a configurable user scheduler for Gnome.

When's Web Page

Copyright (c) 2015 Francesco Garosi

This program comes with ABSOLUTELY NO WARRANTY;
for details, visit http://opensource.org/licenses/bsd-license.php

Close

The purpose of this small utility is to provide the user, possibly without administrative credentials, the ability to define conditions that do not only depend on time, but also on a particular state of the session (e.g. the result of a command run in a shell). The same result could be achieved with scripts that periodically run commands, check the results and react accordingly, but such a simple task could result in complex sets of scripts and settings that would be harder to maintain. **When** was born out of need: I have been a (happy) Ubuntu user for years now, and couldn't think of having a different desktop environment than the one provided by Ubuntu. In *14.04 LTS* (the release I'm using now) the environment is consistent and pleasant. One thing I've noticed that has evolved in Windows is the *Task Scheduler*: in fact it looks more useful and usable than the usual *cron* daemon, at least because it allows some more options to schedule tasks than just the system time. I needed such an utility to perform some file synchronizations when the workstation is idle, and decided to write my own task scheduler targeted to Ubuntu. The scheduler runs in the background, and displays an indicator applet icon for user interaction.

It is not generally intended as a replacement to cron and the Gnome Task Scheduler, although to some extent these utilities might overlap. **When** is intended to be more flexible, although less precise, and to provide an alternative to more complicated solutions – such as the implementation of *cron* jobs that check for a particular condition and execute commands when the condition is verified. In such spirit, **When** is not as fine-grained in terms of doing things on a strict time schedule: the **When** approach is that "*when* a certain condition is met, *then* something has to be done". The condition is checked periodically, and the countermeasure is taken *subsequently* in a relaxed fashion – this means that it might not occur *immediately* in most cases. In fact and with the default configuration, the delay could also consist of a couple of minutes in the worst case.

CHAPTER 2

Installation

When supports several installation types. The easiest way to install it is using one of the packages provided for Ubuntu, that may also be suitable for other Debian based Linux distributions. However, if a different setup is needed (for instance in a *per-user* based installation), it is possible to install **When** directly from a source archive or from a clone of the *Git* repository.

This chapter covers the installation process and the additional actions that should be performed to get **When** up and running for an user. Information is also provided on how to remove the applet from the system or for an user.

Requirements

For the applet to function and before unpacking it to the destination directory, make sure that *Python 3.x*, *PyGObject* for *Python 3.x* and the `xprintidle` utility are installed. Optionally, to enable file and directory monitoring, the `pyinotify` package can be installed. For example, not all of these are installed by default on Ubuntu: in this case the following commands can be used.

```
$ sudo apt-get install python3-gi
$ sudo apt-get install xprintidle
$ sudo apt-get install gir1.2-appindicator3-0.1
$ sudo apt-get install python3-pyinotify
```

The `gir1.2-appindicator3-0.1` package may not be needed on all systems, but some Linux distributions do not install it by default. `python3-pyinotify` is normally considered *optional* but it is mandatory to enable conditions based on changes to the file system.[1]

After the requirements have been fulfilled, the methods below can be used to set up the applet.

[1] Package based installations depend on this: the installation fails if it is not installed.

Package Based Install

As said above, a package provides the quickest and easiest way to have a working installation of **When**. Packages are provided for Ubuntu, although they might work (at least partially) with other Debian based Linux distributions. **When** packages come in two flavors:

1. `when-command`: this is a LSB structured package, especially suitable for Ubuntu and derivatives, that installs the applet in a way similar to other standard Ubuntu packages. The actual file name has the form `when-command_VERSIONSPEC-N_all.deb` where `VERSIONSPEC` is a version specification, and `N` is a number. Pros of this package are mostly that it blends with the rest of the operating environment and that the `when-command` command-line utility is available on the system path by default. Cons are that this setup may conflict with environments that are very different from Ubuntu.
2. `when-command-opt`: this version installs **When** in `/opt/when-command`, and should be suitable for `.deb` based distributions that differ from Ubuntu. The advantage of this method is that the applet is installed separately from the rest of the operating environment and does not clutter the host system. The main drawback is that the `when-command` utility is not in the system path by default and, unless the *PATH* variable is modified, it has to be invoked using the full path, that is as `/opt/when-command/when-command`. The package file name has the form: `when-command-opt-VERSIONSPEC.deb`.

To install a downloaded package, run

```
sudo dpkg --install when-command_VERSIONSPEC-N_all.deb
```

or

```
sudo dpkg --install when-command-opt-VERSIONSPEC.deb
```

depending on the chosen version. After installation, each user who desires to run **When** has to launch `when-command --install` (or `/opt/when-command/when-command --install` if the second method was chosen) in order to find the applet icon in *Dash* and to be able to set it up as a startup application (via the *Settings* dialog box).[2][3]

Warning: The two package types are seen as different by *apt* and *dpkg*: this means that one package type will not be installed *over* the other. When switching package type, the old package *must* be uninstalled before. This also yields when upgrading from packages up to release *0.9.1*, however removal of user data and desktop shortcuts is not required. After a package type switch or an upgrade from release *0.9.1* or older, `when-command --install` should be invoked again, using the full path to the command if appropriate.

Install from a PPA

It is possible to install **When** on recent Ubuntu series from a PPA. This has the advantage of automatically resolving dependencies and to directly set up a fairly stable release with the recommended layout, and to let the user automatically update the software in the ordinary way.

To add the repository, you can simply issue

```
$ sudo add-apt-repository ppa:franzg/when-command
```

and accept to import the related key. Then refresh the packages and install the applet by running

[2] The first method is the preferred one, and it is the one usually referred to throughout the documentation: `when-command` is considered to be in the path, and in the examples and instructions is invoked directly, omitting the full path prefix.

[3] Although an autostart entry is created, it remains inactive by default if the configuration is not modified in the applet settings.

```
$ sudo apt-get update
$ sudo apt-get install when-command
```

from the command line. The other common methods of setting up a PPA using the *Software & Updates* page in *System Settings* and the *Ubuntu Software Center* also work.

Running `when-command --install` is still needed for each user to add **When** to the desktop when installing for the first time.

Install from the Source

A source archive or a *Git* clone can be used to install the package in a directory of choice, but some additional operations are required. However this can be done almost mechanically. In the following example we will suppose that the source has been downloaded in the form of a `when-command-master.zip` archive located in `~/Downloads`, and that the user wants to install **When** in `~/Applications/When`. The required steps are the below:

```
$ cd ~/Applications
$ unzip ~/Downloads/when-command-master.zip
$ mv when-command-master When
$ cd When
$ rm -Rf po scripts .temp .git* setup.* MANIFEST.in stdeb.*
$ for x in applications doc icons man ; do rm -Rf share/$x ; done
$ chmod a+x share/when-command/when-command.py
$ ln -s share/when-command/when-command.py when-command
$ $HOME/Applications/When/when-command --install
```

The `rm` and `for` statements are **not** mandatory: they are only used to remove stuff that is not used by the installed applet and to avoid a cluttered setup. Also, with this installation method, **When** can only be invoked from the command line using the full path (`$HOME/Applications/When/when-command` in the example): to use the `when-command` shortcut, `$HOME/Applications/When` has to be included in the *PATH* variable in `.bashrc`. This means for instance that the creation of a *symbolic link* in a directory already in the user path can cause malfunctions to **When** on command line invocation.[4]

This installation method is useful in several cases: it can be used for testing purposes (it can supersede an existing installation, using the `--install` switch with the appropriate script), to run the applet directly from a cloned repository or to restrict installation to a single user.

The `--install` Switch

When will try to recognize the way it has been set up the first time it's invoked: the `--install` switch creates the desktop entries and icons for each user that opts in to use the applet, as well as the required directories that **When** needs to run correctly and an active autostart entry, that is:

- `~/.config/when-command` where it will store all configuration
- `~/.local/share/when-command` where it stores resources and logs (in the `log` subdirectory).

Note that the full path to the command has to be used on the first run if the `/opt` based package or the manual installation were chosen: in this way **When** can recognize the installation type and set up the icons and shortcuts properly.

[4] A `.tar.gz` archive is provided along with packaged releases, which is the result of a source-based installation: it extracts all the required files in a directory named `When`, but the extraction directory can be moved before the `when-command --install` step.

Removal

When can be uninstalled via `apt-get remove when-command` or `apt-get remove when-command-opt` if a package distribution was used, or by deleting the newly created applet directory (`~/Applications/When` in the above example) if the source was unpacked from an archive or cloned from *Git*.

Also, desktop shortcut symbolic links can be removed as follows:

```
$ rm -f ~/.local/share/applications/when-command.desktop
$ rm -f ~/.config/autostart/when-command-startup.desktop
```

while the following commands can be used to remove applet data and an extra CLI link (if present):

```
$ rm -f ~/.local/bin/when-command
$ rm -Rf ~/.local/share/when-command
$ rm -Rf ~/.config/when-command
```

where the last line can be skipped if **When** is presumed to be reinstalled at a later time.[5]

Of course it has to be shut down before, for example by killing it via `when-command --kill`.

Note: Removal of user data is *not required* when switching package type or changing installation style, provided that the newly installed `when-command` is invoked with the `--install` switch before using the applet. If user data is removed, all *tasks* and *conditions* and other items will have to be recreated from scratch after reinstalling, unless an *export file* exists.

[5] Not all `rm` operations shown here will actually have effect: the instructions follow the most generic case, and some of the files listed for deletion could be missing.

CHAPTER 3

User Manual

Overview

When deals mainly with two types of entities, *Tasks* and *Conditions*. These entity types are used to define *items* using the applet configuration dialog boxes. Unlike similar applications, since *items* do not only represent checks or commands, they are referenced by *name* throughout the user interface of the applet. Item names must:

- begin with either a letter or a number
- contain only letters, numbers, underscores or dashes (no spaces)

and are case sensitive. A *Task* consists of a command, which will run in the default shell, along with an environment and some hints on how to consider it successful or not. *Tasks* correspond to a single command line each: this means that they can consist of several shell commands each, as long as they can fit on a single shell line – for instance using shell separators (such as semicolons, `;`) or other conjunctions or disjunctions. A *Condition* consists of an inherent *test* or associated *event*, a set of *Tasks* and instructions on how tasks should run in case the test succeeds or the event occurs. There is no concept of *failure* in conditions: a condition can either occur or not.

The relationship between *Tasks* and *Conditions* consists in the possibility for a condition to trigger a task: if the tests or events that determine the condition are positive, then the tasks associated with that condition are *very likely* to run – execution of a task may be prevented either by the **When** applet itself, if a previously run task in the same condition occurrence[1] has failed (or succeeded) and the condition is set to break in such an event, or by the underlying system if there is something wrong with the task itself. The latter case is however normally interpreted as a task *failure*.

More than one task can be associated to a condition: if tasks in a set that is associated to a condition can be considered as *independent*, then the user can choose to run the tasks simultaneously in multiple threads[2] otherwise such tasks can run in sequence, and the sequence can be interrupted, at user's choice, either on first task failure or on first task success.

This shows how a single *condition* can be associated to multiple *tasks*. However a *task* can be "reused" in more than one condition: this is particularly useful when a certain action is required to be triggered under different circumstances

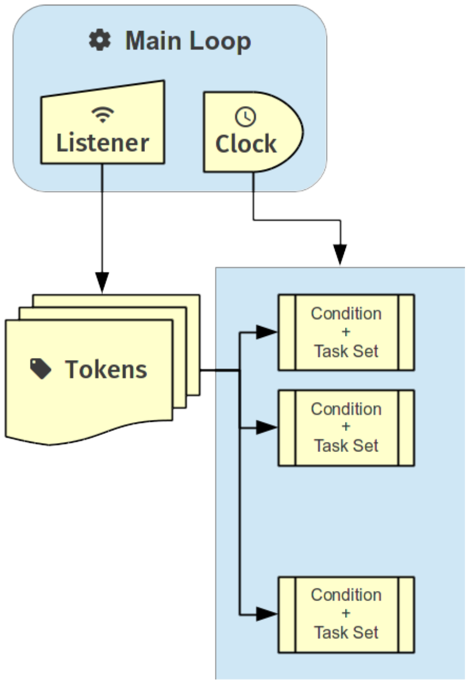
[1] Here a *condition occurrence* refers to an instant in time when the condition prerequisites are verified and, in case of success, the associated task set is scheduled to run, either immediately or shortly after.

[2] There is a limit nevertheless in the number of tasks that can be simultaneously executed, but this limit can be increased in the applet *settings*.

– say, at certain times *and* when an event occurs – in which case both involved conditions may trigger the same task, possibly within different task sets.

Note: Most of the *tokens* that decide whether or not a condition is verified are *enqueued* to be checked at intervals, which in the case of relatively "inexpensive" checks is at every clock tick, while in other cases is at longer intervals defined in the `skip seconds` configuration parameter. For example, conditions that depend on outcome of *external commands* are checked at longer intervals. Also, most[3] system and session events cause the associated condition to be verified at the next clock tick instead of immediately. If such events occur *again* when the task set for the associated condition is already enqueued to be run, it is executed only once – further attempts to enqueue it are simply skipped.

The **When** applet can be thought of as a main loop that incorporates a *clock* and an *event listener*. Whenever the clock ticks, conditions are evaluated, and when an event is caught a *token* is issued for the associated condition to evaluate to *true*, so that the task set that it surrounds can be run. Conditions that do not receive a token or whose test evaluates to *false* are skipped for further evaluation.



Conditions can also be marked as *recurring*: if a condition has *not* been instructed to *repeat checks*, the corresponding tests (and received tokens) are skipped after the first time the associated task set has been triggered, until the **When** applet is restarted – either in a new session or by direct intervention. Enabling the *repeat checks* feature, on the other side, allows the condition to be *recurring*, or even *periodic* in the case of interval based conditions.

There is another type of entity that can be defined, for which the naming convention is the same as for *Tasks* and *Conditions*, that is *Signal Handlers*: these can be used to define special *events* to be caught by *Conditions* when certain *DBus Signals* are emitted. This advanced feature is intended for users with a background on *DBus* specification and is not for general use.[4]

[3] Most events are *deferred*, although there are some whose associated conditions are immediately evaluated: *startup*, *shutdown*, and *suspend* events will cause the respective conditions to immediately trigger their task sets. This choice was necessary because it is virtually impossible to defer events that should occur when the system is shutting down or being suspended, and because the user might expect that tasks that should occur at session startup should be run as soon as possible. The only other type of condition that are validated immediatly on event occurrences are the *command-line* enabled ones that are forced to do so via the `-r` (or `--run-condition`) switch.

[4] This is an advanced feature and is not available by default. It has to be enabled in the program settings to be accessible. Refer to the appropriate chapter for more information.

The Applet

The applet will show up in the indicator tray at startup, which would normally occur at login if the user chose to add **When** to the startup applications. It will read its configuration and start the scheduler in an unattended fashion. Whenever one of the user defined conditions is met, the associated tasks are executed. A small alarm clock icon will display in the indicator tray, to show that the applet is running: by default it turns to an attention sign when the applet requires attention. Also, the icon changes its shape when the applet is *paused* (the clock is crossed by a slash) and when a configuration dialog is open (the alarm clock shows a plus sign inside the circle).

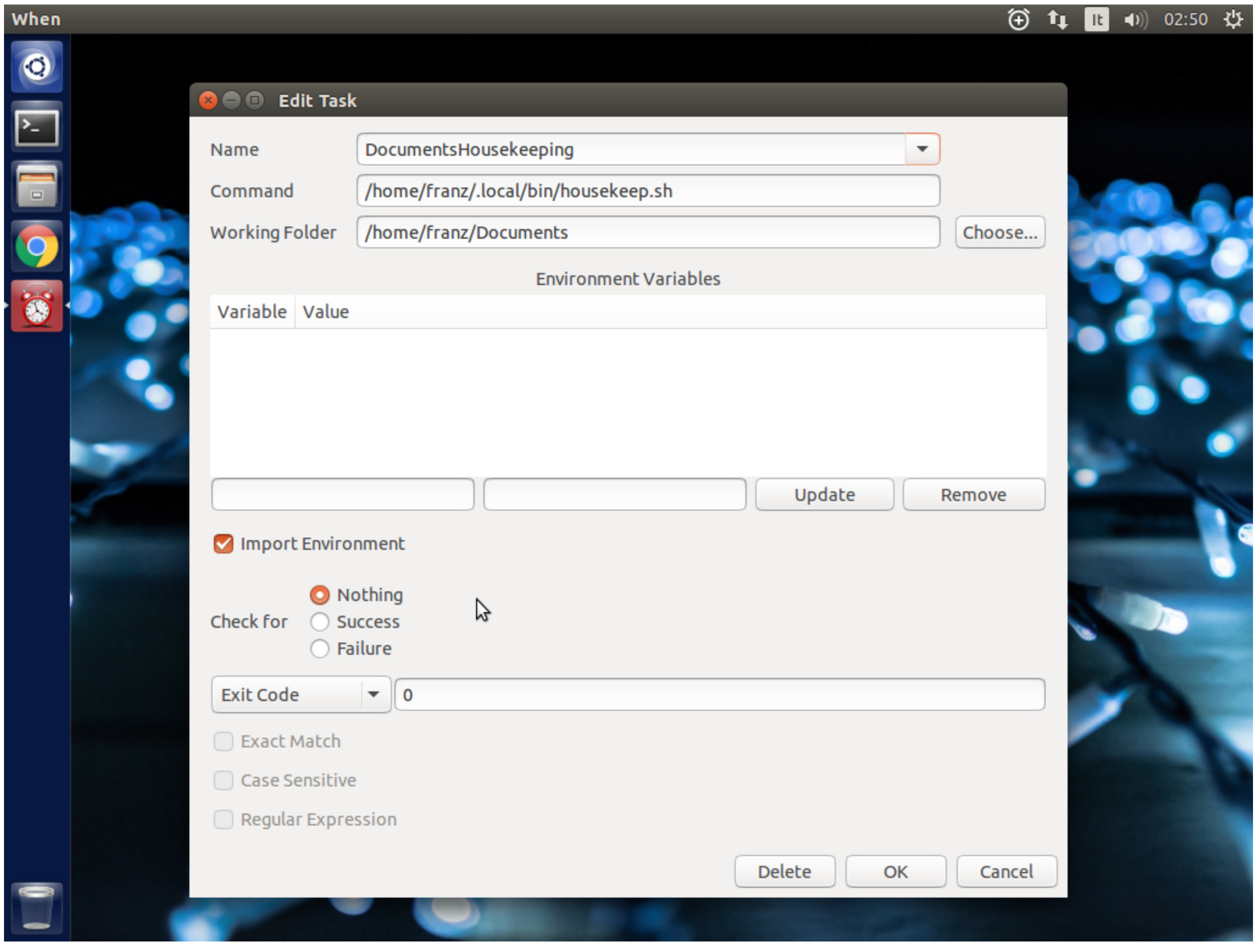
The icon grants access to the main menu, which allows the following basic operations:

- open the *Task* editing dialog box
- open the *Condition* editing dialog box
- open the *Settings* dialog box
- show the Task History window
- *pause* and *resume* the scheduler
- reset condition tests
- show the *About* box
- quit the applet.

Where the *Task* and *Condition* editing boxes, the *Settings* dialog and the *Task History* window might need some more detailed explanation, the other operations should be pretty straightforward: the *Pause* entry pauses the scheduler (preventing any condition to occur), *About...* shows information about the applet and *Quit* shuts the applet down, removing the icon from the top panel.

Some useful features can also be accessed from the *Command Line Interface*, including advanced tools: by default, when the applet is invoked with no arguments, it just starts an instance showing the icon in the top panel (if configured to do so), while the *CLI* allows operations on either a running instance or the applet configuration. The following paragraphs illustrate the details of the applet user interface.

Tasks



Name: DocumentsHousekeeping
Command: /home/franz/.local/bin/housekeep.sh
Working Folder: /home/franz/Documents

Environment Variables

Variable	Value

[✓] Import Environment

Check for: [✓ Nothing] Success Failure

Exit Code: 0

[ ] Exact Match
[ ] Case Sensitive
[ ] Regular Expression

Tasks are basically commands associated with an environment and checks to determine whether the execution was successful or not. The interface lets the user configure some basic parameters (such as the startup directory and the *environment*) as well as what to test after execution (*exit code*, *stdout* or *stderr*). The user can choose to look for the specified text within the output and error streams (when *Exact Match* is unchecked, otherwise the entire output is matched against the given value) and to perform a case sensitive test, or to match a regular expression. In case a regular expression is chosen, the applet will try to search *stdout* or *stderr* for the given pattern. In case of regular expressions, when *Exact Match* is chosen, a match test is performed at the beginning of the output text. Regular expression tests can be case insensitive as well.

The environment in which the subprocess is run can either import the current one (at **When** startup time), use its own variables or both.

The selected task (if any) can be deleted clicking the *Delete* button in the dialog box. However the application will refuse to delete a task that is used in a condition: remove the task reference from the condition first. Every task must have an *unique name*, if a task is named as an existing task it will replace the existing one. The name *must* begin with an alphanumeric character (letter or digit) followed by alphanumerics, dashes and underscores.

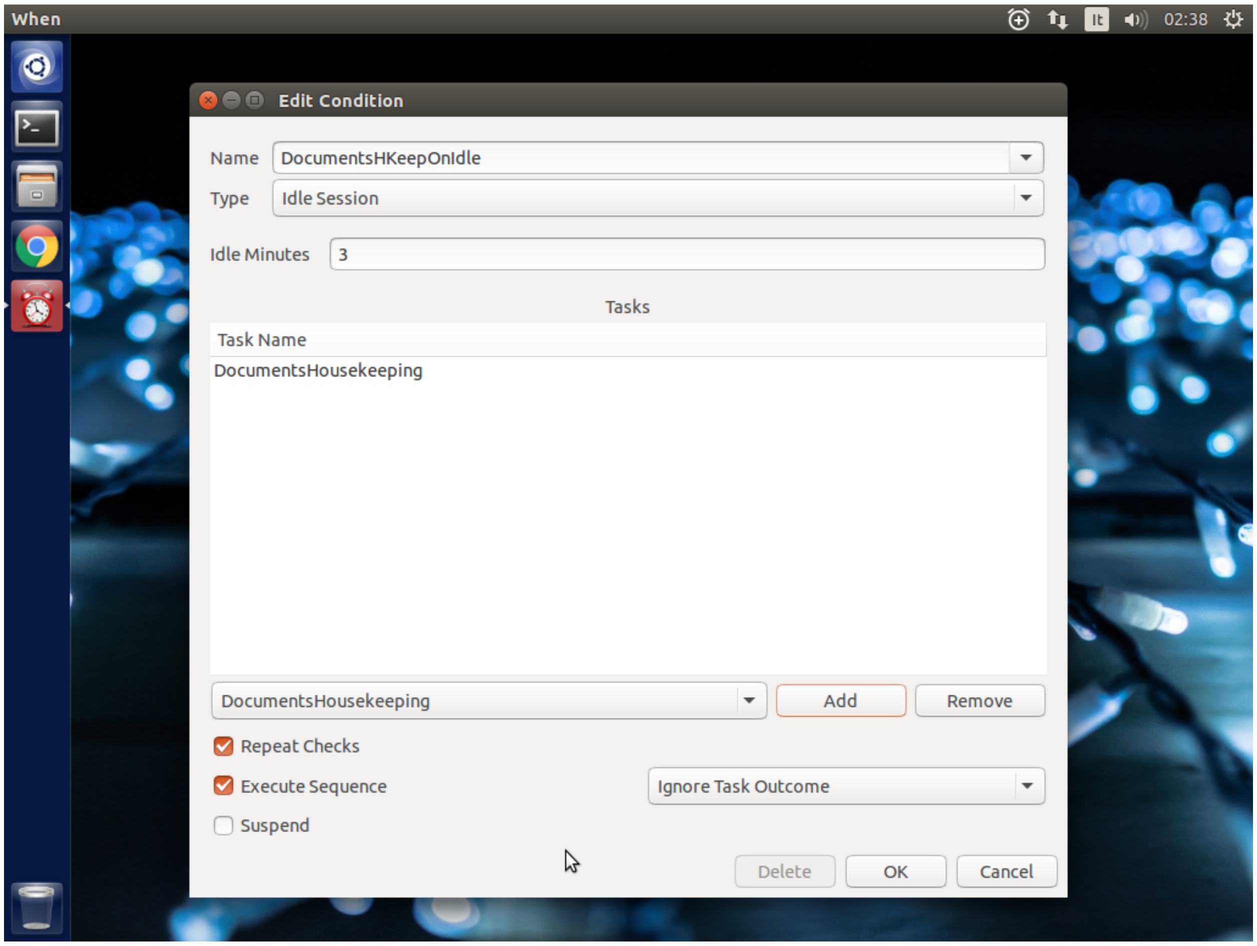
How to use the "Check for" option: The applet can either ignore whatever the underlying process returns to the caller by specifying *Nothing* in the *Check for* group, or check

- exit code
- process output (*stdout*)

- process written errors (*stderr*)

to determine whether the process succeeded or failed. When the user chooses to check for *Success*, the operation is considered successful *if and only if* the process result (exit code, output, or error) corresponds to the user provided value. Same yields for *Failure*: if *Failure* is chosen, only the provided result will indicate a failure. For example, in the most common case the user will choose to expect *Success* to correspond to an *Exit Code* of `0` (in fact the default choice), all other exit codes will indicate a failure. And if the user chooses to expect *Failure* to be reported as the word `Error` in the error messages, whatever other error messages will be ignored and the operation will turn out successful. Please note that since all commands are executed in the default shell, expect an exit code different from `0` when the command is not found. With the `/bin/sh` shell used on Linux, the *not found* code is `127`.

Conditions



Name: DocumentsHKeepOnIdle
Type: Idle Session
Idle Minutes: 3
Tasks
Task Name
DocumentsHousekeeping
DocumentsHousekeeping

Repeat Checks [✓]
Execute Sequence [✓] Ignore Task Outcome
Suspend [ ]

There are several types of condition available:

1. **Interval based:** After a certain time interval the associated tasks are executed, if the condition is set to repeat checks, the tasks will be executed again regularly after the same time interval.
2. **Time based:** The tasks are executed when the time specification is matched. Time definitions can be partial, and in that case only the defined parts will be taken into account for checking: for instance, if the user only specifies minutes, the condition is verified at the specified minute for every hour if the *Repeat Checks* option is set.
3. **Command based:** When the execution of a specified command gives the expected result (in terms of **exit code**,

stdout or **stderr**), the tasks are executed. The way the test command is specified is similar (although simpler) to the specification of a command in the *Task* definition dialog box. The command is run in the same environment (and startup directory) as **When** at the moment it was started.

4. **Idle time based:** When the session has been idle for the specified amount of time the tasks are executed.
5. **Event based:** The tasks are executed when a certain session or system event occurs. The following events are supported:
   - *Startup* and *Shutdown*. These are verified when the applet (or session, if the applet is launched at startup) starts or quits.
   - *Suspend* and *Resume*, respectively match system suspension/hibernation and resume from a suspended state.
   - *Session Lock* and *Unlock*, that occur when the screen is locked or unlocked.
   - *Screensaver*, both entering the screen saver state and exiting from it.
   - *Storage Device Connect* and *Disconnect*, which take place when the user attaches or respectively detaches a removable storage device.
   - *Join* or *Leave a Network*, these are verified whenever a network is joined or lost respectively.
   - *Battery Charging*, *Discharging* or *Low*, respectively occurring when the power cord is plugged, unplugged or the battery is dangerously low: note that a *change* in power status has to arise for the condition to occur, and the *Low* condition is originated from the system.
   - *Command Line Trigger* is a special event type, that is triggered invoking the command line. The associated condition can be scheduled to be run at the next clock tick or immediately using the appropriate switch.
6. **Based on filesystem changes:** The tasks are run when a certain file changes, or when the contents of a directory or its subdirectories change, depending on what the user chose to watch – either a file or a directory. A dialog box can be used to select what has to be watched.[5]
7. **Based on an user defined event:** The user can monitor system events by listening to *DBus* signals emitted on either the system bus or the session bus.[4]

Also, the condition configuration interface allows to decide:

- whether or not to repeat checks even after a task set has been executed – that is, make an action *recurring*;
- to run the tasks in a task set concurrently or sequentially: when tasks are set to run sequentially, the user can choose to ignore the outcome of tasks or to break the sequence on the first failure or success by selecting the appropriate entry in the box on the right – tasks that don't check for success or failure will *never* stop a sequence;
- to *suspend* the condition: it will not be tested, but it's kept in the system and remains inactive until the *Suspend* box is unchecked.

The selected condition (if any) can be deleted clicking the *Delete* button in the dialog box. Every condition must have an *unique name*, if a condition is named as an existing one it will replace it. The name *must* begin with an alphanumeric character (letter or digit) followed by alphanumerics, dashes and underscores.

Note:

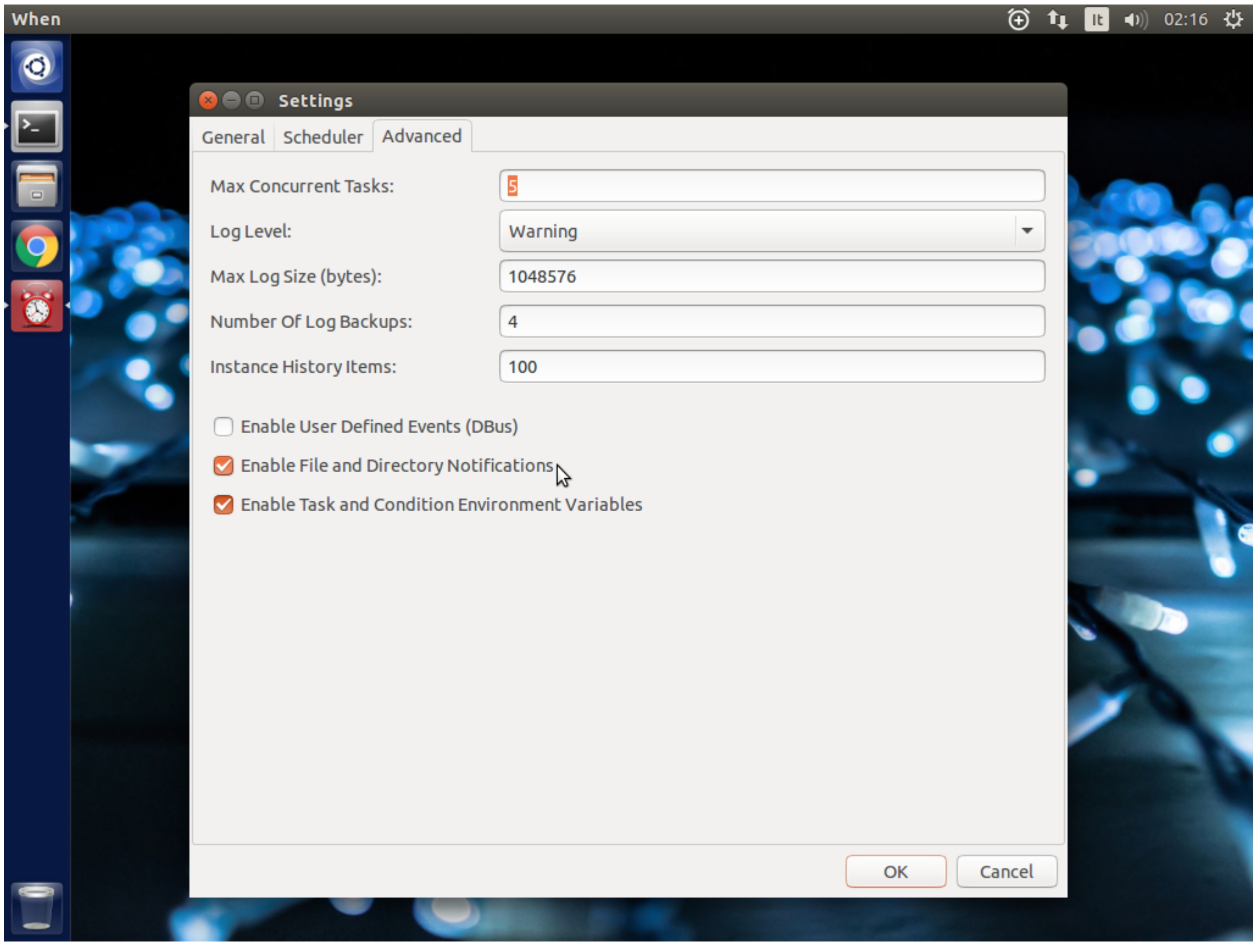
- **Shutdown Conditions.** Because of the way applications are notified that the session is ending (first a `TERM` signal is sent, then a `KILL` if the first was unsuccessful), the *Shutdown* event is not suitable for long running tasks, such as file synchronizations, disk cleanup and similar actions. The system usually concedes a "grace time" of about one second before shutting everything down. Longer running tasks will be run if the users quits

[5] This is an optional feature, and could lack on some systems: to enable it the `pyinotify` library must be installed, refer to the instructions below.

the applet through the menu, though. Same yields for *Suspend*: by specification, no more than one second is available for tasks to complete.

- **Disabled Events.** Some events may not be supported on every platform, even on different Ubuntu implementations. *Screen Lock/Unlock* for instance does not follow very strict specifications, and could be disabled on some desktops. Thus one or more events might appear as *[disabled]* in the list: the user still can choose to create a condition based on a disabled event, but the corresponding tasks will never be run.

Configuration



Max Concurrent Tasks: 5
Log Level: Warning
Max Log Size (bytes): 1048576
Number Of Log Backups: 4
Instance History Items: 100

☐ Enable User Defined Events (DBus)
☑ Enable File and Directory Notifications
☑ Enable Task and Condition Environment Variables

The program settings are available through the specific *Settings* dialog box, and can be manually set in the main configuration file, which can be found in `~/.config/when-command/when-command.conf`.

The options are:

1. **General**

- *Show Icon*: whether or not to show the indicator icon and menu
- *Autostart*: set up the applet to run automatically at login
- *Notifications*: whether or not to show notifications upon task failure
- *Minimalistic Mode*: disable menu entries for item definition dialog boxes and in part reduce memory footprint

- *Icon Theme*: *Guess* to let the application decide, otherwise one of *Dark* (light icons for dark themes), *Light* (dark icons for light themes), and *Color* for colored icons that should be visible on all themes.

2. **Scheduler**

- *Application Clock Tick Time*: represents the tick frequency of the application clock, sort of a heartbeat, each tick verifies whether or not a condition has to be checked and detects if conditions that depend on external events have been already enqueued and are ready to trigger tasks; this option is called `tick seconds` in the configuration file
- *Condition Check Skip Time*: conditions that require some "effort" (mainly the ones that depend on an external command) will skip this amount of seconds from previous check to perform an actual test, should be at least the same as *Application Clock Tick Time*; this is named `skip seconds` in the configuration file
- *Preserve Pause Across Sessions*: if *true* (the default) the scheduler will remain paused upon applet restart if it was paused when the applet (or session) was closed. Please notice that the indicator icon gives feedback anyway about the paused/non-paused state. Use `preserve pause` in the configuration file
- *Reset Condition Tests on Wakeup Events*: automatically restore condition checks for non recurring conditions also on wakeup (usually from suspended state) as if the applet were restarted. The option is `wakeup reset` in the configuration.

3. **Advanced**

- *Max Concurrent Tasks*: maximum number of tasks that can be run in a parallel run (`max threads` in the configuration file)
- *Log Level*: the amount of detail in the log file
- *Max Log Size*: max size (in bytes) for the log file
- *Number Of Log Backups*: number of backup log files (older ones are erased)
- *Instance History Items*: max number of tasks in the event list (*History* window); this option is named `max items` in the configuration file
- *Enable User Defined Events*: if set, then the user can define events using DBus *(see below)*. Please note that if there are any user defined events already present, this option remains set and will not be modifiable. It corresponds to `user events` in the configuration file. Also, to make this option effective and to enable user defined events in the *Conditions* dialog box, the applet must be restarted
- *Enable File and Directory Notifications*: if set, **When** is configured to enable conditions based on file and directory changes. The option may result disabled if the required optional libraries are not installed. When the setting changes, the corresponding events and conditions are enabled or disabled at next startup.
- *Enable Task and Condition Environment Variables*: whether or not to export specific environment variables with task and condition names when spawning subprocesses (either in *Tasks* or in *Command Based Conditions*). The configuration entry is `environment vars`.

The configuration is *immediately stored upon confirmation* to the configuration file, although some settings (such as *Notifications*) might require a restart of the applet. The configuration file can be edited with a standard text editor, and it follows some conventions common to most configuration files. The sections in the file might slightly differ from the tabs in the *Settings* dialog, but the entries are easily recognizable.

By default the applet creates a file with the following configuration, which should be suitable for most setups:

```
[Scheduler]
tick seconds = 15
skip seconds = 60
preserve pause = true
wakeup reset = true
```

```
[General]
show icon = true
autostart = false
notifications = true
log level = warning
icon theme = guess
user events = false
file notifications = false
environment vars = true
minimalistic mode = false

[Concurrency]
max threads = 5

[History]
max items = 100
log size = 1048576
log backups = 4
```

Manual configuration can be particularly useful to bring back the program icon once the user decided to hide it[6] losing access to the menu, by setting the `show icon` entry to `true`. Another way to force access to the *Settings* dialog box when the icon is hidden is to invoke the applet from the command line using the `--show-settings` (or `-s`) switch when an instance is running.

Minimalistic Mode

There is the possibility to start **When** in *Minimalistic Mode* checking the appropriate option in the *General* tab of the *Setting* dialog box. This option is useful mainly when all necessary items are already defined (or the user chooses to define them through *Item Definition Files*, see the *Advanced* guide) and there is no more need to clutter the GUI with "useless" menu entries. This mode has also the side effect of saving some memory, although not a very big amount, by avoiding to load dialog boxes that will not be shown.

The remaining menu entries are:

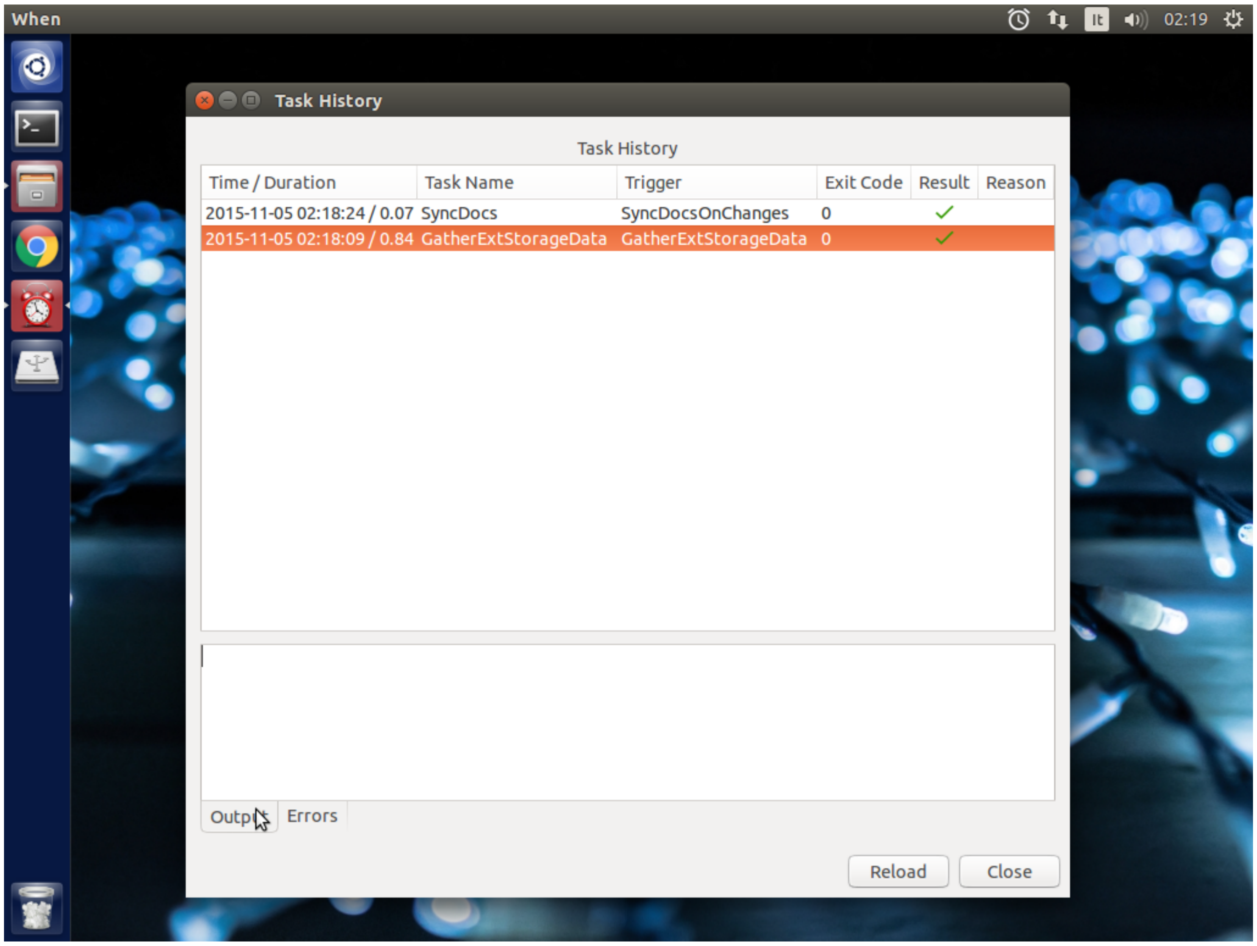
- Settings...
- Pause
- About...
- Quit

which can be useful to revert behavior to normal.

To effectively enter or leave *Minimalistic Mode* the applet must be restarted after the option was changed.

[6] I was doubtful about providing the option, then just decided to implement it and provide a safety net anyway.

The History Window



Time / Duration	Task Name	Trigger	Exit Code	Result	Reason
2015-11-05 02:18:24 / 0.07	SyncDocs	SyncDocsOnChanges	0	✓	
2015-11-05 02:18:09 / 0.84	GatherExtStorageData	GatherExtStorageData	0	✓	

Since logs aren't always user friendly, **When** provides an easier interface to verify task results. Tasks failures are also notified graphically via the attention-sign icon and badge notifications, however more precise information can be found in the *History* box. This shows a list of the most recent tasks that have been launched by the running instance (the list length can be configured), which reports:

- The start time of the task and its duration in seconds
- The task *unique name*
- The *unique name* of the condition that triggered the task
- The process *exit code* (as captured by the shell)
- The result (green *tick mark* for success, red *cross mark* for failure)
- A short hint on the failure *reason* (only in case of failure)

and when the user clicks a line in the table, the tabbed box below will possibly show the output (*stdout*) and errors (*stderr*) reported by the underlying process. The contents of the list can also be exported to a text file, by invoking the applet with the `--export-history` switch from a console window when an instance is running. The file contains exactly the same values as the history list, with the addition of a row identifier at the beginning of the row. Start time and duration are separate values. The first row of the file consists of column mnemonic titles and the value separator is a semicolon: the file can be safely imported in spreadsheets, but column conversions could be needed depending on your locale settings.

Reset Condition Tests

As seen in the paragraph describing the *Conditions* definition dialog box, some conditions can be defined as *non-recurring*: this means that if the test has been successful once in the current **When** session it will be skipped ever since until the applet is restarted. In some cases it may be required by the user that such events are tested again – for example during an unexpectedly long session. In this case it is possible to reset the applet, either using the *Command Line Interface* as explained below, or using the appropriate entry in the menu.

It is also possible to configure the applet to automatically reset the tests for *non-recurring* events in case of a system wakeup by setting the appropriate scheduler options in the *Configuration* dialog. Currently this only supports wakeup from suspended state: this is particularly useful for notebook users that just close the lid to end a session, de facto hibernating the PC.

Command Line Interface

This paragraph illustrates the command line options that can be used to either control the behaviour of a running **When** instance or to handle its configuration or persistent state – consisting of *tasks*, *conditions* and *signal handlers*. Some of the options are especially useful to recover when something has gone the wrong way – such as the `--show-settings` switch mentioned above, or the `-I` (or `--show-icon`) switch, to recover from an unwantedly hidden icon. There are also switches that grant access to “advanced” features, which are better covered in the next sections.

The available options are:

-s, --show-settings	show the settings dialog box of an existing instance, it requires a running instance, which may be queried using the `--query` switch explained below
-l, --show-history	show the history dialog box of an existing instance
-t, --show-tasks	show the task dialog box of an existing instance
-c, --show-conditions	show the condition dialog box of an existing instance
-d, --show-signals	show the DBus signal handler editor box for an existing instance[4]
-R, --reset-config	reset applet configuration to default, requires the applet to be shut down with an appropriate switch
-E, --restart-conditions	reset conditions to be checked as if they had not been already successful: it allows to restore checks also for conditions that are not recurrent
-I, --show-icon	show applet icon
-T, --install	install or reinstall application icon and autostart icon, requires applet to be shut down with an appropriate switch
-C, --clear	clear current tasks, conditions and possibly signal handlers, requires applet to be shut down with an appropriate switch
-Q, --query	query for an existing instance (returns a zero exit status if an instance is running, nonzero otherwise, and prints an human-readable message if the `--verbose` switch is also specified)
-H file, --export-history file	export the current task history (the ones shown in the history box) to the file specified as argument in a CSV-like format
-r cond, --run-condition cond	trigger a command-line associated condition and immediately run the associated tasks; *cond* must be specified and has to be one of the *Command Line Trigger* conditions, otherwise the command will fail and no task will be run

-f cond, --defer-condition cond schedule a command-line associated condition to run the associated tasks at the next clock tick; the same as above yields for *cond*

--shutdown close a running instance performing shutdown tasks first

--kill close a running instance abruptly, no shutdown tasks are run

--item-add file add items from a specially formatted file (see the *advanced* section for details); if the specified file is `-` the text is read from the standard input

--item-del itemspec delete the item specified by *itemspec*. *itemspec* has the form `[type:]item` where `type:` is optional and is is one of `tasks`, `conditions` and `sighandlers` (or an abbreviation thereof) while `item` is the name of an item; `type` can only be omitted if the name is unique

--item-list type print the list of currently managed items to the console, each prefixed with its type; `type` is optional (see above for possible values) and if specified only items of that type are listed

--export file save tasks, conditions and other items to a portable format; the *file* argument is optional, and if not specified the applet tries to save these items to a default file in `~/.config/when-command`; this will especially be useful in cases where the compatibility of the "running" versions of tasks and conditions (which are a binary format) could be broken across releases

--import file clear tasks, conditions and other items and import them from a previously saved file; the *file* argument is optional, and if not specified the applet tries to import these items from the default file in the `~/.config/when-command` directory; the applet has to be shut down before attempting to import items.

Some trivial switches are also available:

-h, --help show a brief help message and exit

-V, --version show applet version, if `--verbose` is specified it also shows the *About Box* of a running instance, if present

-v, --verbose show output for some options; normally the applet would not display any output to the terminal unless `-v` is specified, the only exceptions being `--item-list` that lists all known *items* to the standard output and `--version` that prints out the version string anyway.

Please note that whenever a command line option is given, the applet will not "stay resident" if there is no running instance. On the other side, if the user invokes the applet when already running, the new instance will bail out with an error.

CHAPTER 4

Advanced Features

This chapter describes the advanced features of **When**. Some of these can be handy for everyday use too, others may require some deeper insight in what occurs under the hood in a desktop session. Anyway these features can be enabled or disabled in the applet settings, and can be safely ignored if not needed.

File and Directory Notifications

Monitoring file and directory changes can be enabled in the *Settings* dialog box. This is particularly useful to perform tasks such as file synchronizations and backups, but since file monitoring can be resource consuming, the option is disabled by default. File and directory monitoring is quite basic in **When**: a condition can be triggered by changes either on a file or on a directory, no filter can be specified for the change type – that is, all change types are monitored (creations, writes and deeletions), and in case of directory monitoring all files in the directory are recursively monitored. These limitations are intentional, at least for the moment, in order to keep the applet as simple as possible. Also, no more than either a file or a directory can be monitored by a condition: in order to monitor more items, multiple conditions must be specified.

Warning: As said above, this feature is optional: to make it available the `pyinotify` package has to be installed. The Ubuntu package manager can handle it (`sudo apt-get install python3-pyinotify`); alternatively `pip` can be used to let Python install it directly: `sudo pip3 install pyinotify` (this will ensure that the latest release is installed, but package updates are left to the user). If **When** is installed via a PPA the package manager will take care to install all the dependencies, including the optional ones.

There are also some configuration steps at system level that might have to be performed if filesystem monitoring does not work properly: when a monitored directory is big enough, the default *inotify watches* may fall short so that not all file changes can be notified. When a directory is under control, all its subdirectories need to be watched as well recursively, and this implies that several watches are consumed. There are many sources of information on how to increase the amount of *inotify watches*, and as usual StackExchange is one of the most valuables: see Kernel inotify watch limit reached for a detailed description. Consider that other applications and utilities, especially the ones that synchronize files across the network – such as the cloud backup and synchronization clients – use watches intensively. In fact **When** monitoring activity should not be too different from other cases.

Conditions depending on file and directory monitoring are not synchronous, and checks occur on the next tick of the applet clock. Depending tasks should be aware that the triggering event might have occurred some time before the notified file or directory change.

DBus Signal Handlers

Recent versions of the applet support the possibility to define system and session events using DBus. Such events can activate conditions which in turn trigger task sequences, just like any other condition. However, since this is not a common use for the **When** scheduler as it assumes a good knowledge of the DBus interprocess communication system and the related tools, this feature is intentionally inaccessible from the applet menu and disabled by default in the configuration. To access the *DBus Signal Handler Editor* dialog, the user **must** invoke the applet from the command line with the appropriate switch, while an instance is running in the same session:

```
$ when-command --show-signals
```

This is actually the only way to expose this dialog box. Unless the user defines one or more signal handlers, there will be no *User Defined Events* in the corresponding box and pane in the *Conditions* dialog box, and **When** will not listen to any other system and session events than the ones available in the *Events* list that can be found in the *Conditions* dialog box. The possibility to define such events must be enabled in the *Settings* dialog box, and **When** has to be restarted to make the option effective: before restart the user events are not available in the *Conditions* box, although it becomes possible to show the *DBus Signal Handler Editor* using the command shown above. If the appropriate setting is disabled, the above command exits without showing the editor dialog.

To define a signal to listen to, the following values must be specified in the *DBus Signal Handler Editor* box:

- the handler name, free for the user to define as long as it begins with an alphanumeric character (letter or digit) followed by alphanumerics, dashes and underscores
- the bus type (either *Session* or *System* bus)
- the unique bus name in dotted form (e.g. `org.freedesktop.DBus`)
- the path of the object that emits the signal (e.g. `/org/freedesktop/FileManager1`)
- the interface name in dotted form (e.g. `org.freedesktop.FileManager1`)
- the signal name
- whether the scheduler must wait until the next clock tick to process the signal (checking *Activate on next clock tick*)

All these values follow a precise syntax, which can be found in the DBus documentation. Moreover, if the signal has any parameters, constraints on the parameters can be specified for the condition to be verified: given a list of constraints, the user can choose whether to require all of them or just any to evaluate to true. The tests against signal parameters require the following data:

- *Value #* is the parameter index
- *Sub #* (optional) is the index within the returned parameter, when it is either a list or a dictionary: in the latter case, the index is read as a string and must match a dictionary key
- *comparison* (consisting of an operator, possibly negated) specifies how the value is compared to a test value: the supported operators are
    1. = (equality): the operands are converted to the same type, and the test is successful when they are identical; notice that, in case of boolean parameters, the only possible comparison is equality (and the related *not* equality): all other comparisons, if used, will evaluate to false and prevent condition activation, and the comparison value should be either *true* or *false*

2. `CONTAINS`: the test evaluates to true when either the test string is a substring of the selected value, or the parameter is a list (or struct, or dictionary: for dictionaries it only searches for values and not for keys though), no *Sub #* has been specified, and the test value is in the compound value
3. `MATCHES`: the test value is treated as a *regular expression* and the selected value, which must be a string, matches it
4. < (less-than): the selected value is less than the test value (converted to the parameter inferred or introspected value type)
5. > (greater-than): the selected value is greater than the test value (converted to the parameter inferred or introspected value type)

- *Test Value* is the user provided value to compare the parameter value to: in most cases it is treated as being of the same type as the selected parameter value.

When all the needed fields for a tests are given, the test can be accepted by clicking the *Update* button. To remove a test line, either specify *Value #* and *Sub #* or select the line to delete, then click the *Remove* button. Tests are optional: if no test is provided, the condition will be enqueued as soon as the signal is emitted. If a test is specified in the wrong way, or a comparison is impossible (e.g. comparing a returned list against a string), or any error arises within a test, the test will evaluate to *false* and the signal will not activate any associated condition. For now the tests are pretty basic: for instance nested compound values (e.g. lists of lists) are not treated by the testing algorithm. The supported parameter types are booleans, strings, numerals, simple arrays, simple structures, and simple dictionaries. Supporting more complex tests is beyond the scope of a limited scheduler: the most common expected case for the DBus signal handler is to catch events that either do not carry parameters or carry minimal information anyway.

Note: When the system or session do not support a bus, path, interface, or signal, the signal handler registration fails: in this case the associated event never takes place and it is impossible for any associated condition to be ever verified.

Environment Variables

By default **When** defines one or two environment variables when it spawns subprocesses, respectively in *command based conditions* and in *tasks*. These variables are:

- `WHEN_COMMAND_TASK` containing the task name
- `WHEN_COMMAND_CONDITION` containing the name of the triggering or current condition

When the test subprocess of a command based condition is run, only `WHEN_COMMAND_CONDITION` is defined, on the other hand when a task is run both are available. This feature can be disabled in the configuration file or in the *Settings* dialog box if the user doesn't want to clutter the environment or the variable names conflict with other ones. Please note that in a *task* these variables are defined *only* if the task is set to import the environment (which is true by default): if not, it will only know the variables defined in the appropriate list.[1]

Item Definition File

In recent releases another way to define *items* (*tasks*, *conditions* and especially *signal handlers*) has been introduced, that uses text files whose syntax is similar (although it differs in some ways) to the one used in common configuration files. Roughly, an *item definition* file has the following format:

[1] This behavior is intentional, since if the user chose not to import the surrounding environment, it means that it's expected to be as clean as possible.

```
[NameOf_Task-01]
type: task
command: do_something
environment variables:
  SOME_VAR=some appropriate value
  ANOTHER_VAR=42
check for: failure, status, 2

[ThisIs_Cond02]
type: condition
based on: file_change
watched path: /home/myaccount/Documents
task names: NameOf_Task-01

[SigHandler_03]
type: signal_handler
bus: session
bus name: org.ayatana.bamf
object path: /org/ayatana/bamf/matcher
interface: org.ayatana.bamf.matcher
signal: RunningApplicationsChanged
parameters:
  0:1, not equal, BoZo

# this is the end of the file.
```

where the names in square brackets are item names, as they appear in the applet dialog boxes. Such names are case sensitive and follow the same rules as the related *Name* entries in dialog boxes: only names that begin with an alphanumeric character and continue with *alphanumerics*, *underscores* and *dashes* (that is, no spaces) are accepted. Entries must be followed by colons and in case of entries that support lists the lists must be indented and span multiple lines. Complex values are rendered using commas to separate sub-values. The value for each entry is considered to be the string beginning with the first non-blank character after the colon.

Warning: Even a single error, be it syntactical or due to other possibly more complex discrepancies, will cause the entire file to be rejected. The loading applet will complain with an error status and, if invoked using the `--verbose` switch, a very brief error message: the actual cause of rejection can normally be found in the log files.

For each item, the item name must be enclosed in square brackets, followed by the entries that define it. An entry that is common to all items is `type`: the type must be one of `task`, `condition` or `signal_handler`. Every other value will be discarded and invalidate the file. The following sections describe the remaining entries that can (or have to) be used in item definitions, for each item type. Entry names must be written in their entirety: abbreviations are not accepted.

Tasks

Tasks are defined by the following entries. Some are mandatory and others are optional: for the optional ones, if omitted, default values are used. Consider that all entries correspond to entries or fields in the *Task Definition Dialog Box* and the corresponding default values are the values that the dialog box shows by default.

- `command`: The value indicates the full command line to be executed when the task is run, it can contain every legal character for a shell command. *This entry is mandatory*: omission invalidates the file.

- `environment variables`: A multi-value entry that includes a variable definition on each line. Each definition has the form `VARNAME=value`, must be indented and the value *must not* contain quotes. Everything after the equal sign is considered part of the value, including spaces. Each line defines a single variable.
- `import environment`: Decide whether or not to import environment for the command that the task runs. Must be either `true` or `false`. Defaults to *true*.
- `startup directory`: Set the *startup directory* for the task to be run. It should be a valid directory.
- `check for`: The value of this entry consists either of the word `nothing` or of a comma-separated list of three values, that is `outcome, source, value` where
  - `outcome` is either `success` or `failure`
  - `source` is one of `status`, `stdout` or `stderr`
  - `value` is a free form string (it can also contain commas), which should be compatible with the value chosen for `source` – this means that in case `status` is chosen it should be a number.

  By default, as in the corresponding dialog box, if this entry is omitted the task will check for success as an exit status of `0`.
- `exact match`: Can be either `true` or false. If `true` in the post-execution check the entire *stdout* or *stderr* will be checked against the *value*, otherwise the value will be sought in the command output. By default it is *false*. It is only taken into account if `check for` is specified and set to either *stdout* or *stderr*.
- `regexp match`: If `true` the value will be treated as a *regular expression*. If also `exact match` is set, then the regular expression is matched at the beginning of the output. By default it is *false*. It is only taken into account if `check for` is specified and set to either *stdout* or *stderr*.
- `case sensitive`: If `true` the comparison will be made in a case sensitive fashion. By default it is *false*. It is only taken into account if `check for` is specified and set to either *stdout* or *stderr*.

Signal Handlers

Signal handlers are an advanced feature, and cannot be defined if they are not enabled in the configuration: read the appropriate section on how to enable *user defined events*. If user events are enabled, the following entries can be used:

- `bus`: This value can only be one of `session` or `system`. It defaults to *session*, so it has to be specified if the actual bus is not in the *session bus*.
- `bus name`: Must hold the *unique bus name* in dotted form, and is *mandatory*.
- `object path`: The path to the objects that can issue the signal to be caught: has a form similar to a *path* and is *mandatory*.
- `interface`: It is the name of the object interface, in dotted form. *Mandatory.*
- `signal`: The name of the signal to listen to. This too is *mandatory*.
- `defer`: If set to `true` (the default), the signal will be caught but the related condition will be fired at the next clock tick instead of immediately.
- `parameters`: This is a multiple line entry, and each parameter check must be specified on a single line. Each check has the form: `idx[:sub], compare, value` where
  - `idx[:sub]` is the parameter index per *DBus* specification, possibly followed by a subindex in case the parameter is a collection. `idx` is always an integer number, while `sub` is an integer if the collection is a list, or a string if the collection is a dictionary. The interpunction sign is a colon if the subindex is present.
  - `compare` is always one of the following tokens: `equal`, `gt`, `lt`, `matches` or `contains`. It can be preceded by the word `not` to negate the comparison.

  - `value` is an arbitrary string (it can also contain commas), without quotes.

- `verify`: Can be either `all` or `any`. If set to `any` (the default) the parameter check evaluates to *true* if any of the provided checks is positive, if set to `all` the check is *true* only if all parameter checks are verified. It is only taken into account if `parameters` are verified.

If user events are not enabled and a signal handler is defined, the item definition file will be invalidated.

Conditions

Conditions are the most complex type of items that can be defined, because of the many types that are supported. Valid entries depend on the type of condition that the file defines. Moreover, *conditions* depend on other items (*tasks* and possibly *signal handlers*) and if such dependencies are not satisfied the related condition – and with it the entire file – will be considered invalid.

The following entries are common to all types of condition:

- `based on`: Determines the type of condition that is being defined. It *must* be one of the following and is *mandatory*:
  - `interval` for conditions based on time intervals
  - `time` for conditions that depend on a time specification
  - `command` if the condition depends on outcome of a command
  - `idle_session` for condition that arise when the session is idle
  - `event` for conditions based on *stock* events
  - `file_change` when file or directory changes trigger the condition
  - `user_event` for conditions arising on user defined events: these can only be used if user events are enabled, otherwise the definition file is discarded.

  Any other value will invalidate the definition file.
- `task names`: A comma separated list of tasks that are executed when the condition fires up. The names *must* be defined, either in the set of existing tasks for the running instance, or among the tasks defined in the file itself.
- `repeat checks`: If set to `false` the condition is never re-checked once it was found positive. By default it is *true*.
- `sequential`: If set to `true` the corresponding tasks are run in sequence, otherwise all tasks will start at the same time. *True* by default.
- `suspended`: The condition will be suspended immediately after construction if this is *true*. *False* by default.
- `break on`: Can be one of `success`, `failure` or `nothing`. In the first case the task sequence will break on first success, in the second case it will break on the first failure. When `nothing` is specified or the entry is omitted, then the task sequence will be executed regardless of task outcomes.

Other entries depend on the values assigned to the `based on` entry.

Interval

Interval based conditions require the following entry to be defined:

- `interval minutes`: An integer *mandatory* value that defines the number of minutes that will occur between checks, or before the first check if the condition is not set to repeat.

Time

All parameters are optional: if none is given, the condition will fire up every day at midnight.

- `year`: Integer value for the year.
- `month`: Integer value for month: must be between 1 and 12 included.
- `day`: Integer value for day: must be between 1 and 31 included.
- `hour`: Integer value for hour: must be between 0 and 23 included.
- `minute`: Integer value for minute: must be between 0 and 59 included.
- `day of week`: A token, one of `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday`, `sunday`. No abbreviations allowed.

Command

Command based conditions accept a command line and the specification of what has to be checked. The latter is not mandatory, and defaults to expectation of a zero exit status.

- `command`: The full command line to run: this is *mandatory*.
- `check for`: Somewhat similar to the same entry found in *Tasks*, this entry must be specified as a comma-separated pair of the form `source, value`, where `source` is one of `status`, `stdout` or `stderr`, and `value` is an integer in the `status` case, or a string to look for in the other cases. Defaults to `status, 0`.
- `match regexp`: If `true` the test value is treated as a *regular expression*. Defaults to `false`.
- `exact match`: If `true` the test value is checked against the full output (if `match regexp` is `true` the regular expression is matched at the beginning of the output). Defaults to `false`.
- `case sensitive`: If `true` the comparison will be case sensitive. Defaults to `false`.

Idle Session

The only parameter is mandatory:

- `idle minutes`: An integer value indicating the number of minutes that the machine must wait in idle state before the condition fires.

Event

This condition type requires a sigle entry to be defined.

- `event type`: This *must* be one of the following words:
  - `startup`
  - `shutdown`
  - `suspend`
  - `resume`
  - `connect_storage`
  - `disconnect_storage`
  - `join_network`

  - `leave_network`
  - `screensaver`
  - `exit_screensaver`
  - `lock`
  - `unlock`
  - `charging`
  - `discharging`
  - `battery_low`
  - `command_line`

Each of them is a single word with underscores for spaces. Abbreviations are not accepted. Any other value invalidates the condition and the file.

File and Path Modifications

Also in this case a single entry is required, indicating the file or path that **When** must observe.

- `watched path`: A path to be watched. Can be either the path to a file or to a directory. No trailing slash is required, but it has to be a full path (it could be relative if the user is sure of where **When** is launched from).

User Event

In this case a single entry is required and must contain the *name* of an user defined event. The event can either be defined in the same file or already known to the applet, but it *must* be defined otherwise the file fails to load. Names, as usual, are case sensitive.

- `event name`: The name of the user defined event.

Note: Items defined in an *items definition file*, just as items built using the applet GUI, will overwrite items of the same type and name.

Exporting and Importing Items

When saves *tasks*, *conditions* and *signal handlers* in binary form for use across sessions. It might be useful to have a more portable format at hand to store these items and be sure, for instance, that they will be loaded correctly when upgrading **When** to a newer release. While every effort will be made to avoid incompatibilities, there might be cases where compatibility cannot be kept.

To export all items to a file, the following command can be used:

```
$ when-command --export [filename.dump]
```

where the file argument is optional. If given, all items will be saved to the specified file, otherwise in a known location in `~/.config`. The saved file is not intended to be edited by the user – it uses a JSON representation of the internal objects.

To import items back to the applet, it has to be shut down first and the following command must be run:

```
$ when-command --import [filename.dump]
```

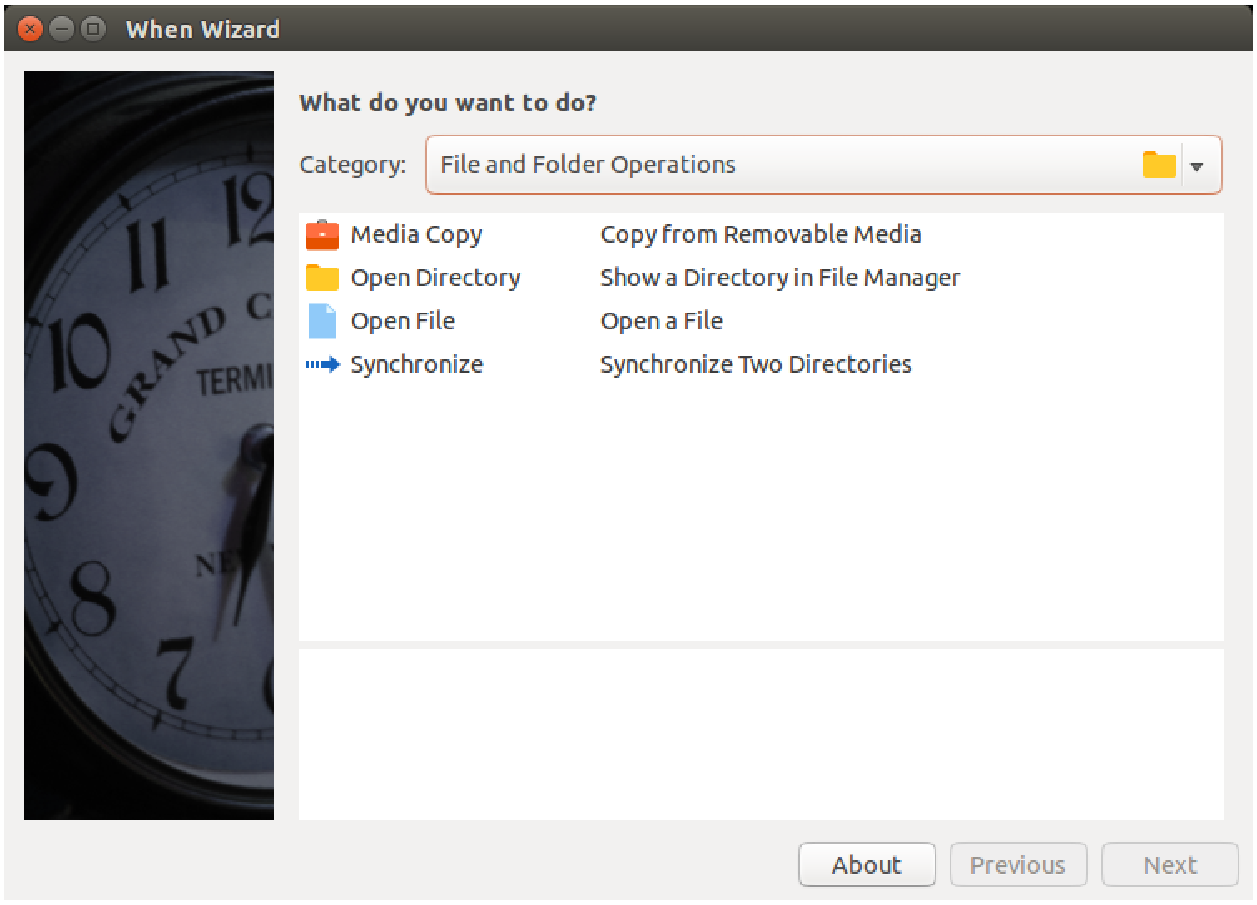
where the `filename.dump` parameter must correspond to a file previously generated using the `--export` switch. If no argument is given, **When** expects that items have been exported giving no file specification to the `--export` switch. After import **When** can be restarted.

CHAPTER 5

The When Wizard

The **When Wizard** is a suite of utilities that aim at providing an easier user interface for a rich subset of **When**‘s capabilities, in order to give to end users with lesser interest in programming and scripting the possibility to use the **When** scheduler to perform simple but useful tasks. In fact, the use of **When** as it is assumes a certain knowledge of the command shell, including its constructs and peculiarities, which is somehow in contrast with the GUI nature of the applet itself.

Warning: This section refers to software in its early development stage: it may contain bugs and errors, and it might be subject to changes in both appearance and functionality. The documentation will be kept as updated as possible, however there might be a gap sometimes between updates to the software and related documentation changes.

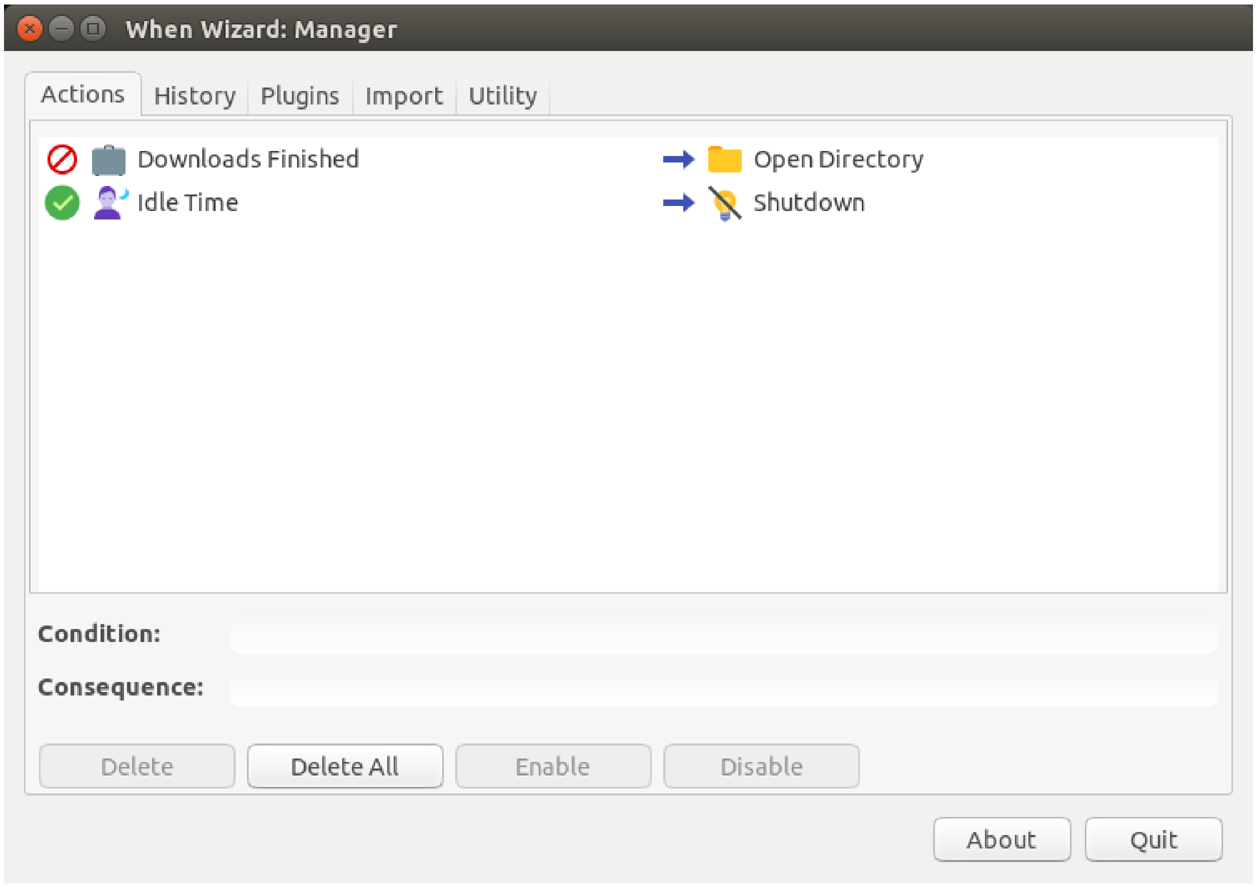


Media Copy	Copy from Removable Media
Open Directory	Show a Directory in File Manager
Open File	Open a File
Synchronize	Synchronize Two Directories

As the name suggests, the main **When Wizard** application presents itself with a *wizard* styled interface, which allows to define what has to be done and the circumstances under which it has to happen in a more intuitive step-by-step fashion. **When** is still used to do most of the job, that is scheduling checks, listening to events and performing tasks when the conditions are met, but the wizard instructs it on how to behave instead of requiring the user to find out what commands and events have to be specified in its low-level interface.[1]

While the wizard interface is used to define tasks and conditions, there is another utility, named the **When Wizard Manager**, which can be used for several tasks, including the removal of *actions* (tasks surrounded by circumstances) that are no more needed, viewing the history of past actions in a simplified way and some environment tuning. Thanks to the modular, extensible nature of the **When Wizard** suite, the manager application can also be used to install plugins for actions that are not available by default in the distribution.

[1] If you want to use *both* types of interface, avoid names beginning with the `00wiz99_` prefix for *tasks*, *conditions* and *signal handlers* when using the **When** base interface: this sequence is used by the wizard to identify its own items. It is otherwise perfectly legal and can be used if you plan to use **When** alone.



This chapter's intent is to give a brief introduction to the **When Wizard** suite: apart from some specific **When Wizard Manager** configuration options the interface is designed to be as intuitive as possible, and the actions are documented in the interface.

Installation

At the moment, there is no proper installation procedure nor distribution specific package. However getting the **When Wizard** to work is quite easy anyway: it is sufficient to either clone the GitHub repository or download and unpack the zip file from the master branch. It may be useful to move the resulting directory (and maybe rename it in case the zip file has been used) in a folder where applications reside. Assuming that the zip file method was chosen and that the `when-wizard-master.zip` file is now in `~/Downloads`:

```
~$ mkdir Apps     # or another name you might like
~$ cd Downloads
~/Downloads$ unzip when-wizard-master.zip
~/Downloads$ mv when-wizard-master ../Apps/when-wizard
~/Downloads$ cd ..
~$
```

or any equivalent combination of operations from the graphical shell. To start the **When Wizard** is now sufficient to

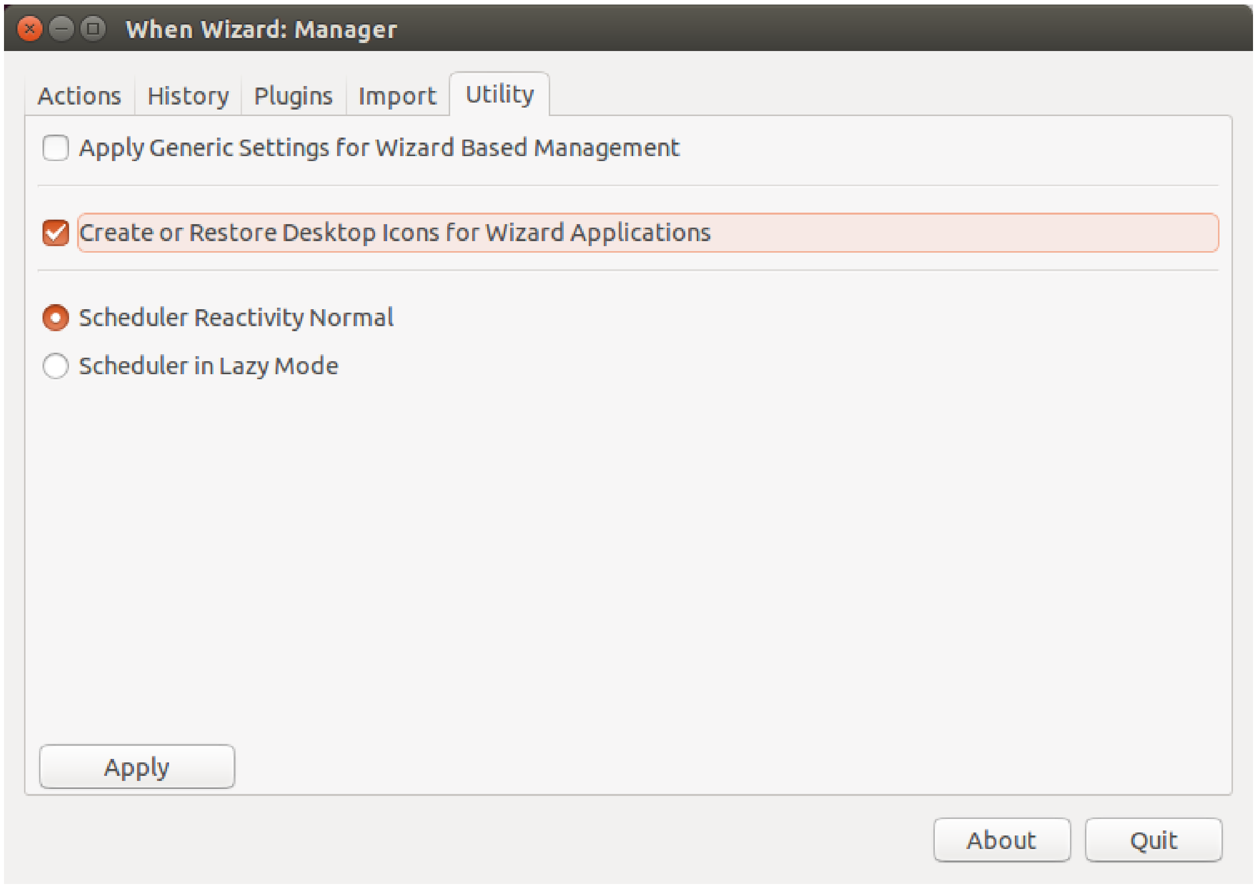
```
~$ cd Apps/when-wizard
~/Apps/when-wizard$ ./when-wizard start-wizard
```

while for the manager application, the subcommand changes:

```
~$ cd Apps/when-wizard
~/Apps/when-wizard$ ./when-wizard start-manager
```

Note: The `when-wizard` command is already marked as *executable* in the repository, however, should the shell refuse to execute the command, it is sufficient to `cd` to the installation directory and issue `chmod a+x when-wizard`; the same yields for all the scripts stored in the `share/when-wizard/scripts` subdirectory.

Using the **When Wizard Manager** it is possible to create icons for both the wizard and the management application: once the **When Wizard Manager** started, choose the *Utility* tab and click on the check box labeled *Create or Restore Icons for Wizard Applications*, then click the *Apply* button:



Apply Generic Settings for Wizard Based Management [ ]

Create or Restore Desktop Icons for Wizard Applications [✓]

[✓ Scheduler Reactivity Normal]

Scheduler in Lazy Mode

The icons for **When Wizard** and for the **When Wizard Manager** should now be available in the *Dash* or whatever menu system is used.

Of course the **When Wizard** suite depends on **When**, and both implicitly and explicitly shares the main application dependencies. If **When** has been installed through a package, be it via the provided PPA or using a downloaded `.deb` bundle, all dependencies for the main **When Wizard** applications should be already satisfied.

However some of the actions that come with the default installation (that is, excluding third party plugins), depend on other packages that are not present, for example, in Ubuntu installations by default. At the moment the required packages, which are however present in the standard Ubuntu repositories, are:

- *mailutils* for the actions that send an e-mail[2]

[2] The mail utilities must be properly configured: when the *mailutils* package is installed, the package manager triggers a configuration page on the console that was used to run `apt-get`. Probably the most likely configuration is the *smart host* based one.

- *consolekit* for power management related actions.

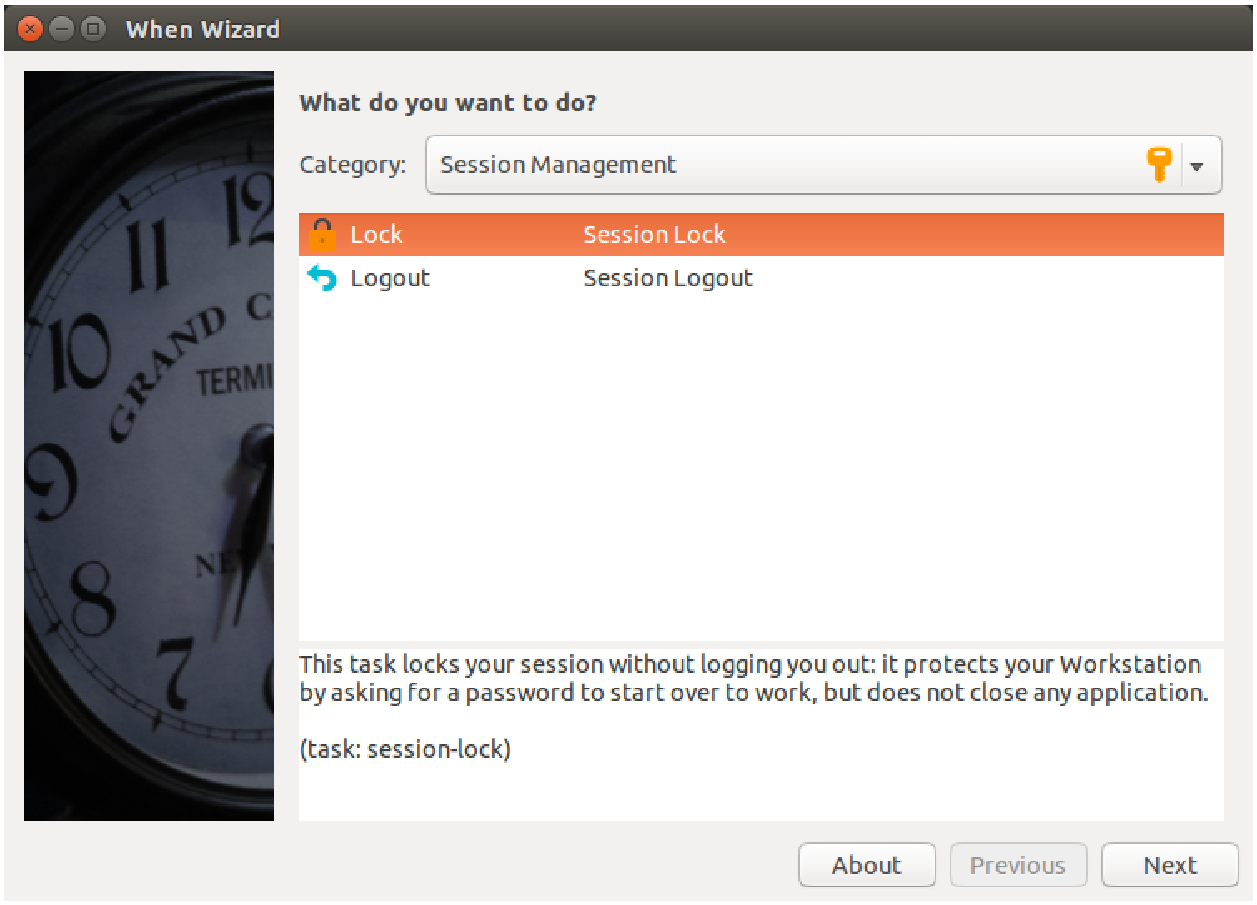
These packages can be easily installed via the usual

```
$ sudo apt-get install mailutils consolekit
```

on Ubuntu and probably its supported derivatives.

Defining Actions

Action definition is simple, but there are many different choices for tasks and surrounding conditions to be described in detail. However, whenever a task or a condition is chosen in the list proposed by the wizard interface, descriptive text is provided along with the name of the item, and in the lower part of the form a longer description appears that better specifies what has been selected:

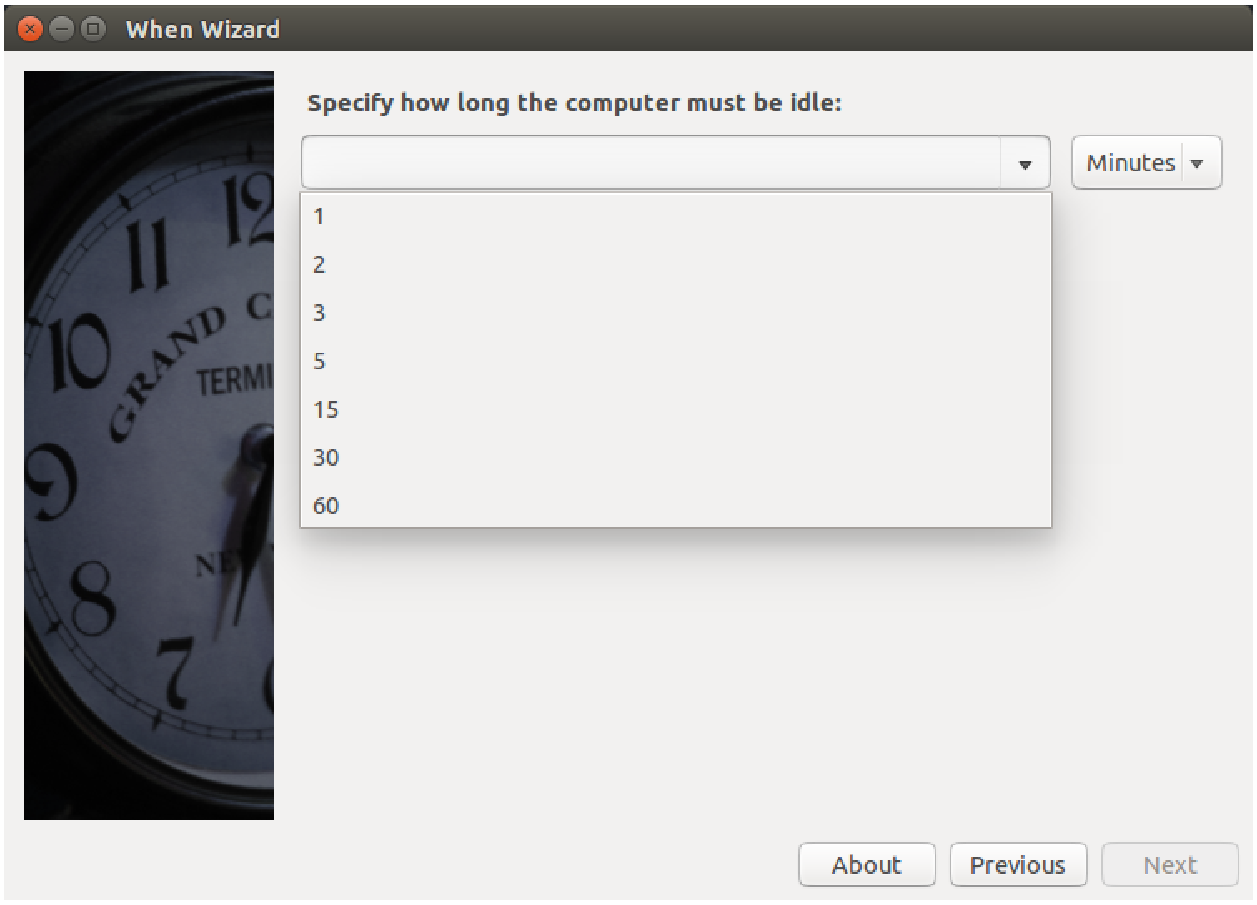


Lock	Session Lock
Logout	Session Logout

This task locks your session without logging you out: it protects your Workstation by asking for a password to start over to work, but does not close any application.

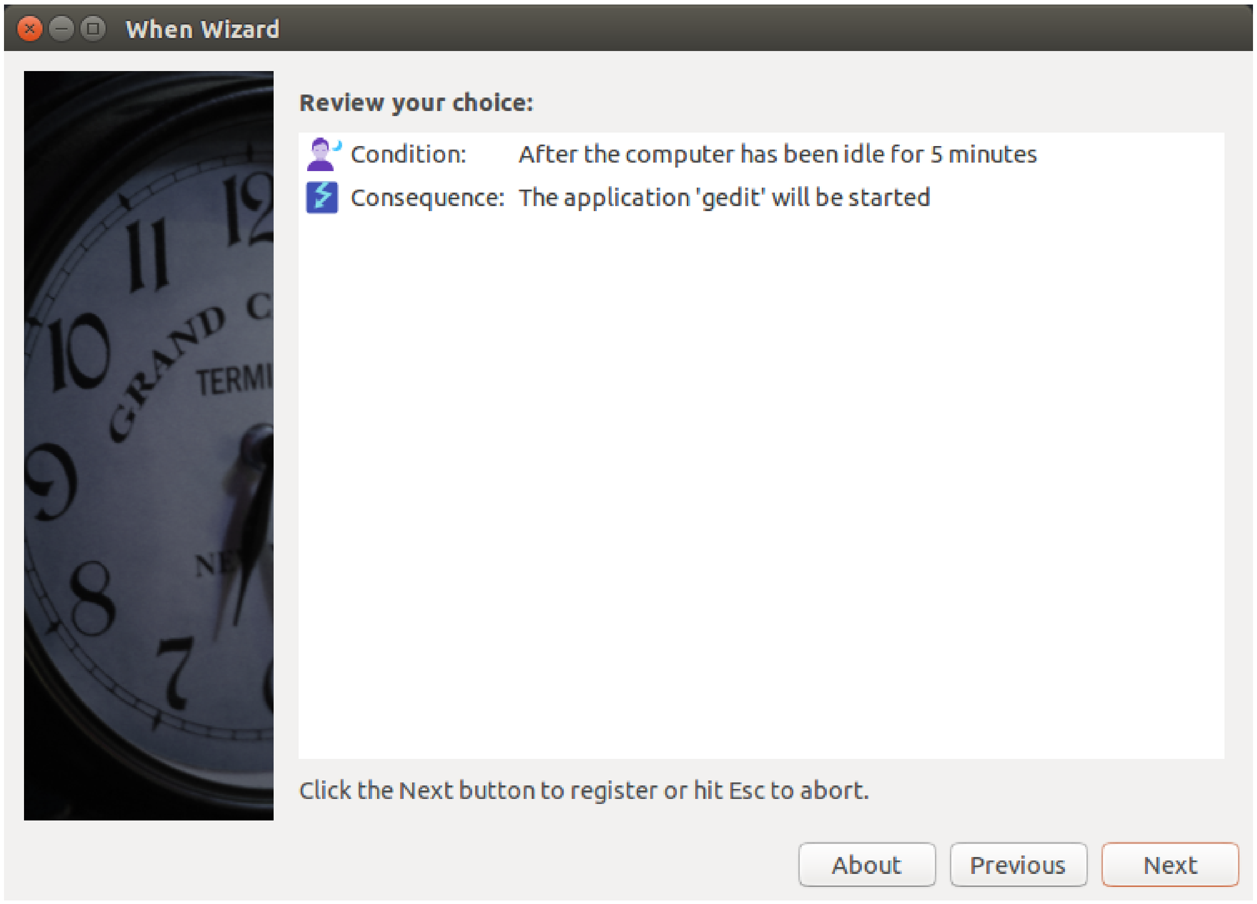
(task: session-lock)

Task and condition items are grouped in categories and types, which can be chosen from the drop-down list at the beginning of the window. The interface proposes to decide what task has to be accomplished and *then* to define the circumstances under which it should happen. Most items have to be configured and after their selection a simple configuration page is presented to the user:



Specify how long the computer must be idle:

The configuration page depends obviously on the selected item. Once the task and circumstance have been chosen and possibly configured, by clicking the *Next* button it is possible to review the action details:



Review your choice:

Condition: After the computer has been idle for 5 minutes

Consequence: The application 'gedit' will be started

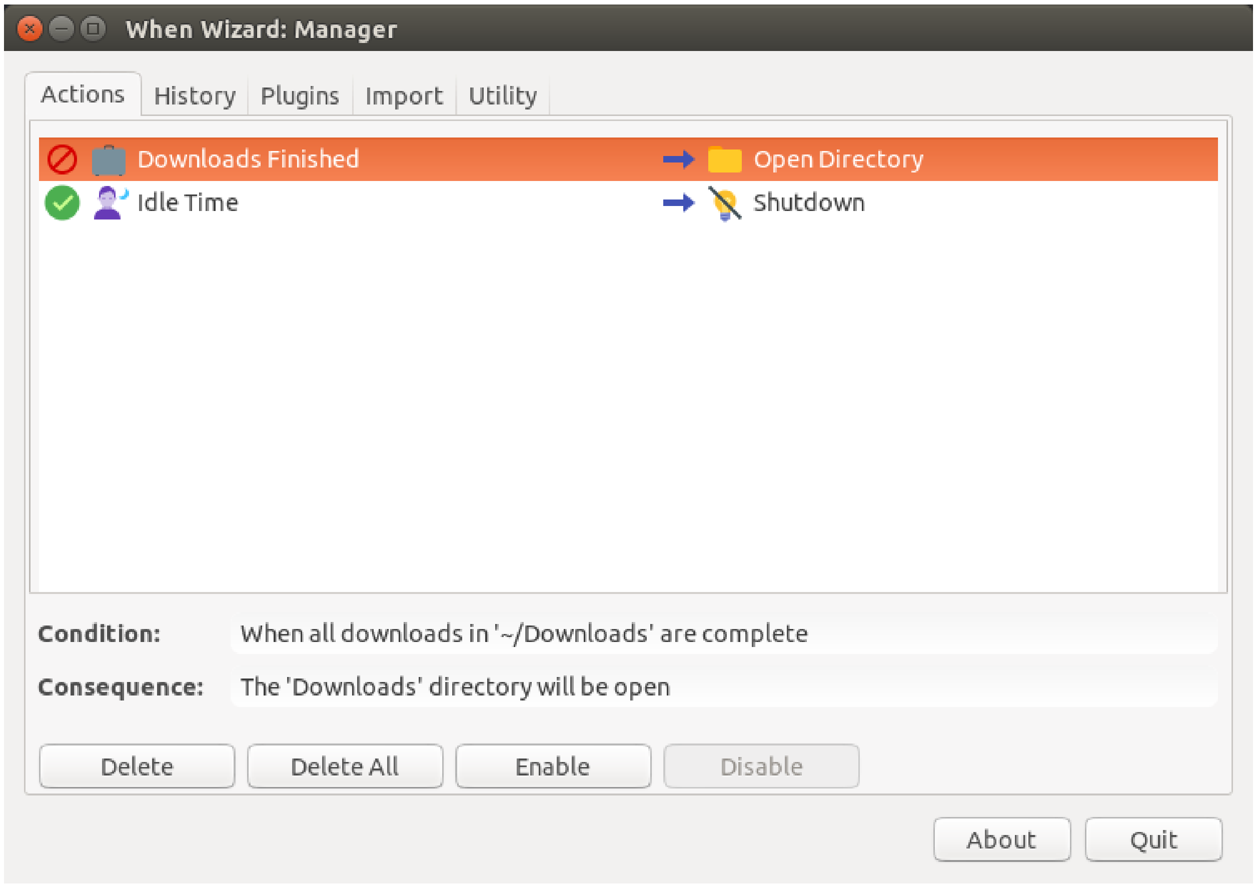
Click the Next button to register or hit Esc to abort.

here the user is still in time to change her or his mind and either modify anything or completely abort the operation by pressing the *Esc* key (or just closing the window). If the *Next* button is clicked, the action is registered in **When**.

Managing Actions

Actions can be removed through the **When Wizard Manager** application. On startup it shows a list containing the actions currently defined using the wizard interface:[3]

[3] Other actions directly defined in **When** are left untouched by the **When Wizard Manager**: of course it is advisable to choose one and only one interface for **When** and avoid its base UI if the wizard approach is chosen, but in this way it is anyway possible to avoid that the wizard interface could mess up a configuration made at lower level, for example using third party *item definition files*.



Downloads Finished → Open Directory
Idle Time → Shutdown

Condition: When all downloads in '~/Downloads' are complete

Consequence: The 'Downloads' directory will be open

To remove an action it is sufficient to select it from the list (a more detailed description is shown under the list) and click the *Delete* button. After confirmation, the action is completely removed from the system.

If the user only wants to suspend checks and consequences for an action, the first page of the manager application also gives the possibility to just *disable* (and possibly *reenable*) a previously defined action: selecting the appropriate line in the list (enabled actions are marked with a green circle containing a tick mark) and clicking the *Disable* button causes the action to be ineffective without deleting it. It can be enabled again at a later time when needed, by just selecting it and clicking the *Enable* button.

Other Uses for the When Wizard Manager

There are some more uses for the manager application, organized in pages:

- visualization of action history
- **When Wizard** plugin management
- third-party provided *item sets* management
- tuning of the underlying **When** scheduler instance.

History visualization is quite trivial: each history record is shown as a line in the visible list, prefixed with startup time and duration of the related action. The tuning and utility page (the one that can be used to create or restore icons too) also does not need a lot of explaination: it just allows to adopt a set of options for **When** that let it better blend with a wizard based usage, including activation of user-defined events and file monitoring, and *Minimalistic Mode* for the applet indicator icon. Settings forced through this page are permanent and can only be reset from the *Settings* dialog box in the main **When** interface. Maybe it's worth to mention that the so-called *lazy mode* is lazy indeed, and in some cases the time between the conditional event and its consequence can be more than six minutes: it is mainly useful when the computer is left alone most of the time while performing tasks (for example: data collection, or very big downloads and so on), while for other configurations *normal reactivity* is possibly the suitable setting – which corresponds to the default values in **When** configuration.

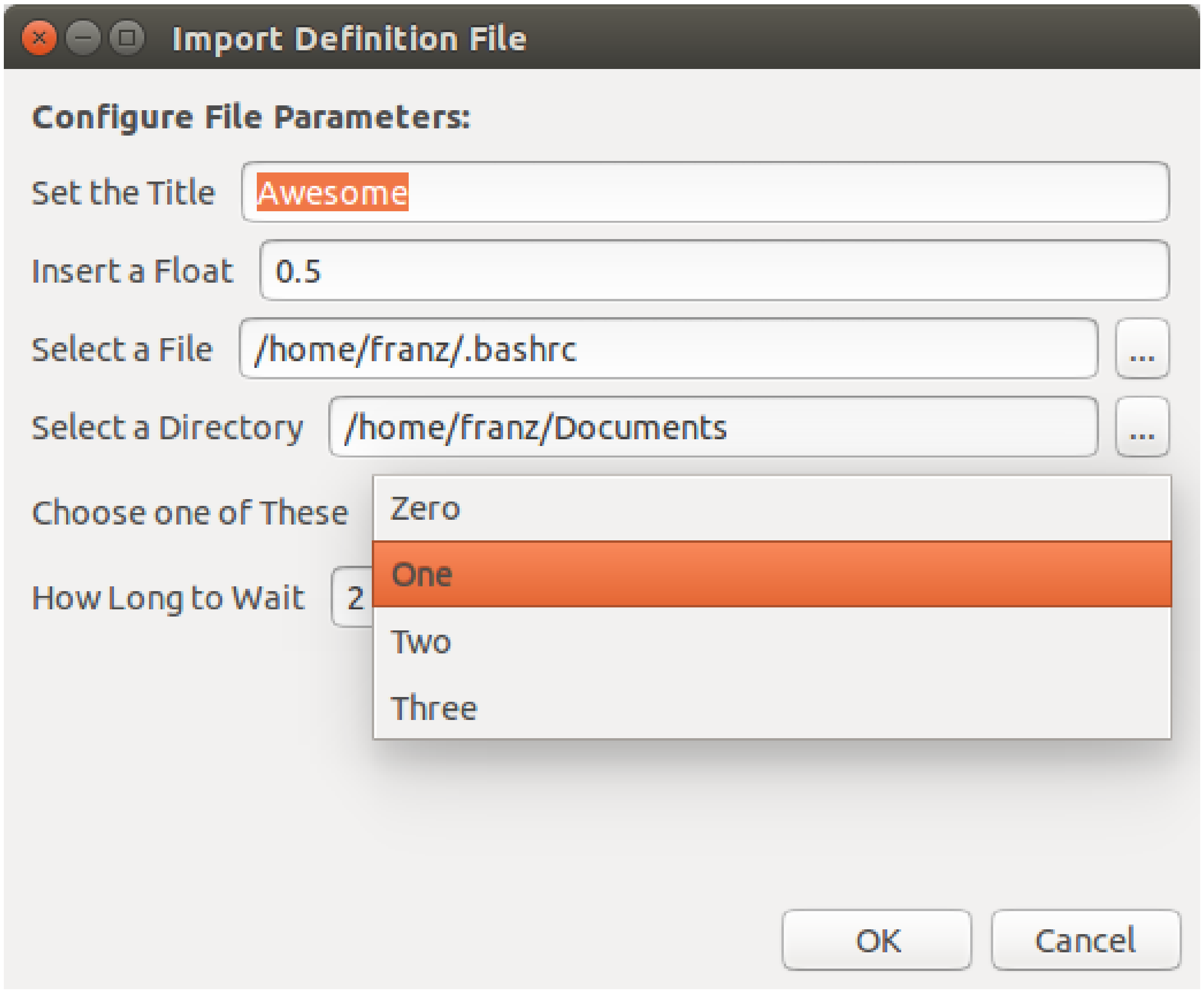
The existence of a *plugin management* page reflects one important aspect of the **When Wizard** application: functionality can be extended through add-ons. Such add-ons (or *plugins*, as they are named in the UI) provide ways to encapsulate common tasks and to grant access to system events and environmental conditions in a simple way: the user might need to configure a small number of options in many cases, and in some cases not even that. *Plugins* can be downloaded in packaged form, and installed and removed from the manager interface.

Warning: Particular care must be taken when installing a plugin: plugins should only be installed from trusted sources much in the same way as software packages. In fact, although plugin code is never run with administrator privileges, a plugin may install scripts that have access to valuable information.

Plugins come packaged with a `.wwpz` extension: if the user writes by himself the path to the package in the appropriate text entry, she or he can use whatever file specification. If the file chooser dialog box is used only files with the `.wwpz` extension will be shown. Plugins can also be removed, but *only if there is no action using them*: to remove a plugin one has to make sure that all related actions have been removed too.

Last but not least, the *Import* page of the **When Wizard Manager** offers the possibility to import preconfigured **When** items via provided *Item Definition Files*. Such files can contain single items as well as item sets, and in fact some third-party defined actions might come packaged in an *Item Definition File*. Files of this type should have a `.widf` extension, but the same considerations yield as for the `.wwpz` files.

In some cases *Item Definition Files* might require some configuration by the user: if so, when the *Execute* button is pressed, a dialog box is shown that gives the possibility to modify some parameters.


Import Definition File

Configure File Parameters:

Set the Title: Awesome

Insert a Float: 0.5

Select a File: /home/franz/.bashrc

Select a Directory: /home/franz/Documents

Choose one of These: Zero / One / Two / Three

How Long to Wait: 2

The parameters should have been documented by who provided the file, and might be subject to checks to verify their correctness at confirmation time.

Some plugins (namely, the ones that depend on user defined events) may require that the user imports an *Item Definition File*. Such cases should be well documented and the developer should provide both the plugin package and the supporting *Item Definition File*.

CHAPTER 6

Tutorial

This tutorial's goal is to provide some *simple* examples to configure the **When** applet to do useful things. Since it is a background application, the examples shown here will focus on operations that would normally take place without user interaction, such as file synchronization, file system housekeeping and so on. However **When** can be used for many other *tasks*, such as automated builds, gathering information, massive file conversions and so on.

Assumptions

When is explicitly aimed at Ubuntu desktops. Probably it would work on other Linux flavors too, and especially on Ubuntu derivatives, maybe exposing full or almost full functionality. However, for the sake of this tutorial, I'll assume that the user is running **When** on *Ubuntu 14.04 LTS* or a more recent release. Within the document the Ubuntu "idioms" will be constantly used, and the examples will favor utilities such as `apt-get` and `dpkg` with respect to the corresponding `yum` and `rpm` of RPM-based distros. *bash* is assumed to be the main shell, and will be used in scripts and interactive shell examples, and administrative tasks will be prefixed by `sudo` as the Ubuntu custom suggests. Packages existing in standard repositories will be referenced by their Ubuntu names. Since many packagers have followed almost the same conventions to organize common packages on Linux systems, it should not be too difficult to find appropriate name conversions online.

I'll also assume that the following packages have been installed:

- `python3-gi`
- `xprintidle`
- `python3-pyinotify`

as suggested in the documentation, as well as the **When** package from the provided `.deb` file, following the suggested "easy" installation method, and that the `when-command` executable is in the path. The applet itself is assumed to be installed in the desktop and initialized for the user via

```
$ when-command --install
```

and that the configuration still has not been modified from the default one provided at initialization time. Long story short, I assume the user to have a working default installation of **When** on a recent Ubuntu desktop.

Support Software

Some examples will use third-party open source software to perform tasks. For example, I wrote **When** to be able to run Unison unattended on idle time. *Unison* is a (beautiful and) useful piece of software, that doesn't come with Ubuntu by default: in some examples the tutorial will also require the user to install and configure *Unison* and other software.

Scripts

In some cases it's easier to group actions in shell scripts, or even to write simple programs, instead of relying on **When**'s sequential task run ability. In such cases, apart from being embedded in the tutorial text, the scripts are available in source form for download.

All possible effort is taken to make the scripts portable without modification, and to make unavoidable modifications as easy as possible. In the latter case, there will be instructions on how to modify the script.

The Examples

The provided examples illustrate simple tasks that could serve as a starting point to build more complex actions. Me too, I started developing **When** with very simple goals in mind. Then **When** ended up including more interesting features on which I also built some tools – one of them is even used to help develop **When** itself. The provided examples are about

- *File Synchronization and Backup*
- *Housekeeping*
- *Automatic Import* from a storage device.

The examples will try to cover the entire process of creating the tasks, their trigger conditions and all the accessory parts and scripts needed for each example to work from scratch. In particular, the tutorial will also try to guide the user through the configuration process when different settings from the default ones are needed.

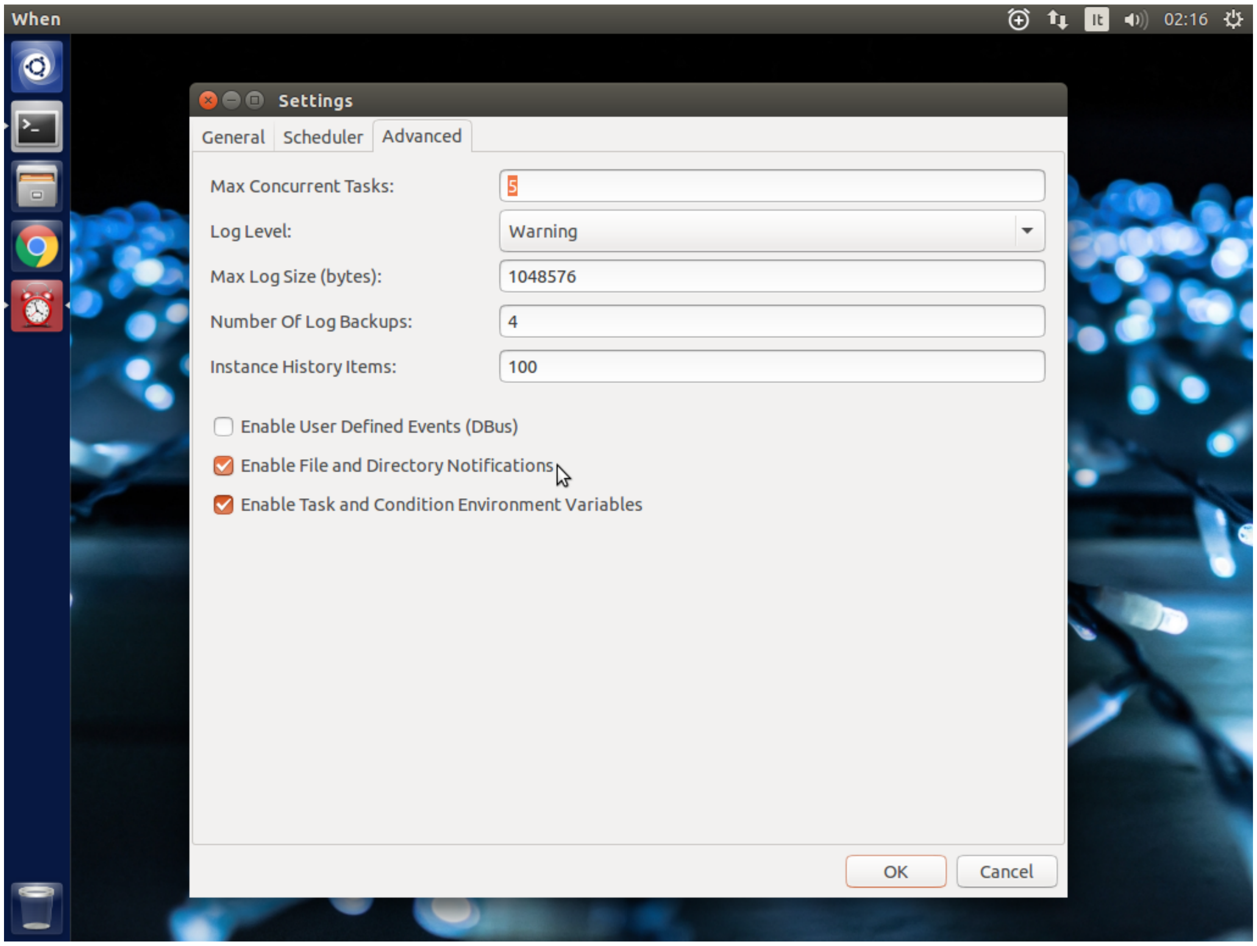
File Synchronization and Backup

The first example illustrates how to synchronize files from a directory to a backup destination. It uses *Unison* to copy the files from the source location to the destination, only in one direction, whenever the contents of the source directory change: in other words it implements a *file change condition* to determine whether or not to backup the source directory. Because such condition checks are always deferred, subsequent writes to a file in the origin directory will not likely trigger the same amount of synchronizations, which is good for backup tasks. Here we will synchronize the `~/Documents` directory with a mounted location. We'll assume there is a NFS disk mounted on `/remote`, with a directory owned by the user and named after the user name. To prepare the environment, a proper backup location has to be created, via

```
$ mkdir -p /remote/$USER/backup/Documents
```

in order to keep the environment as clean as possible. The path above is supposed to act as a backup directory only, and to be left untouched unless in case of need: this ensures that synchronization will always be unidirectional.

For this example to function, the *file change conditions* have to be enabled. To do this, open the settings dialog box by clicking on the applet icon in the top panel and selecting *Settings...*.



Max Concurrent Tasks: 5
Log Level: Warning
Max Log Size (bytes): 1048576
Number Of Log Backups: 4
Instance History Items: 100

☐ Enable User Defined Events (DBus)
☑ Enable File and Directory Notifications
☑ Enable Task and Condition Environment Variables

Then click the *Advanced* tab and check the *Enable File and Directory Notifications* entry. **When** has to be restarted for the option to actually work: choose *Quit* from the applet menu, then start **When** from the *dash*.

Install and Setup Unison

A brief description follows on how to setup a simple profile in *Unison* for the sake of this example. This utility can do much more, refer to the specific documentation for details.

Installation

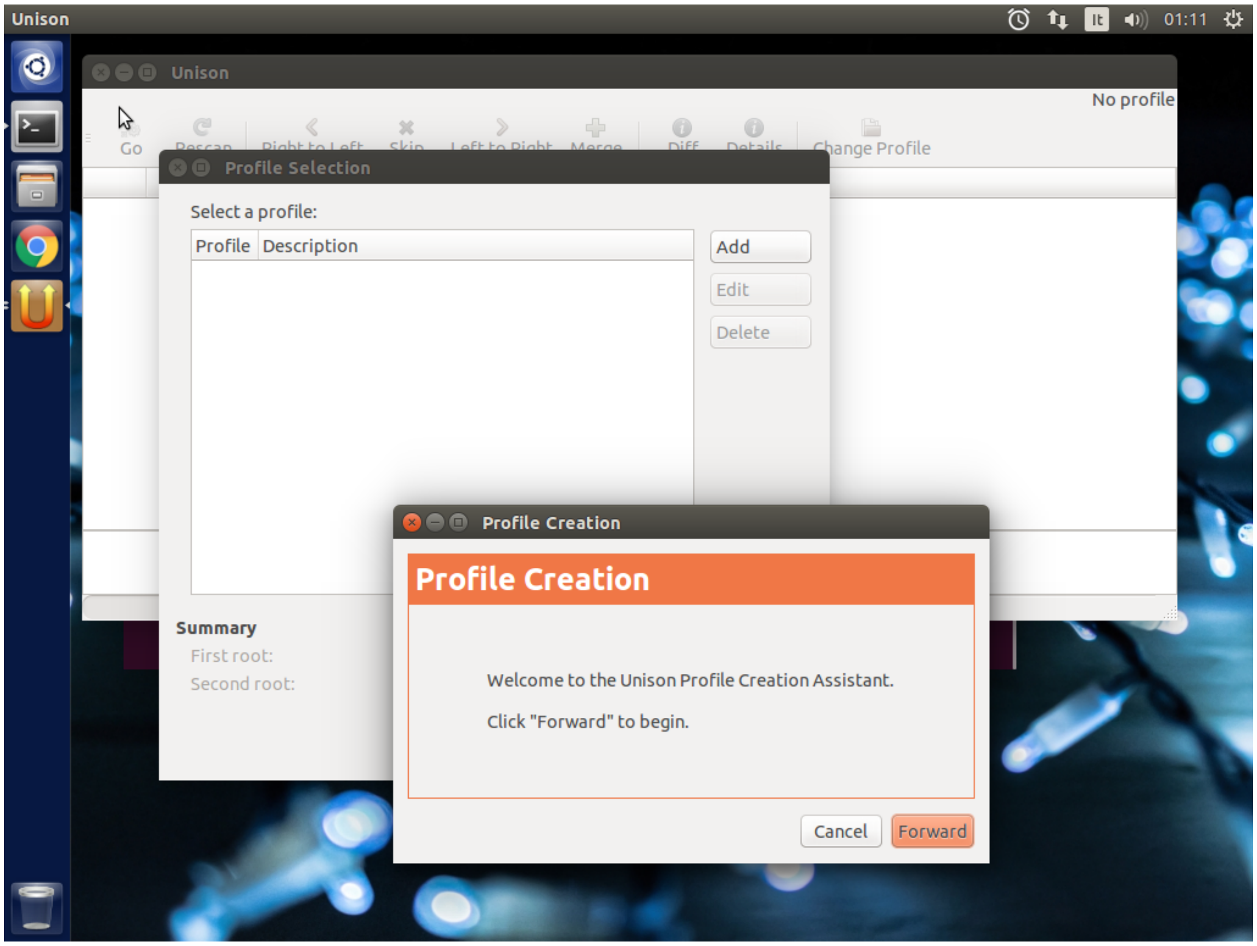
In Ubuntu *Unison* is provided in the official repositories, namely in *Universe*. If that repository is enabled, you can install the software using

```
$ sudo apt-get install -y unison unison-gtk
```

or via the *Ubuntu Software Center*. In this way you can use the *Unison* GUI to configure the synchronization profile: launch *Unison* from *dash* to access its interface (logout and login might be required to find the application in *dash*).

Setup a Synchronization Profile

Open the *Unison* GUI: the main window will be presented, along with a dialog box to select the profile to run. As there are no configured profiles, we will click the *Add* button, and a wizard will help us with the creation of a profile.



Click *Forward* and follow these easy steps:

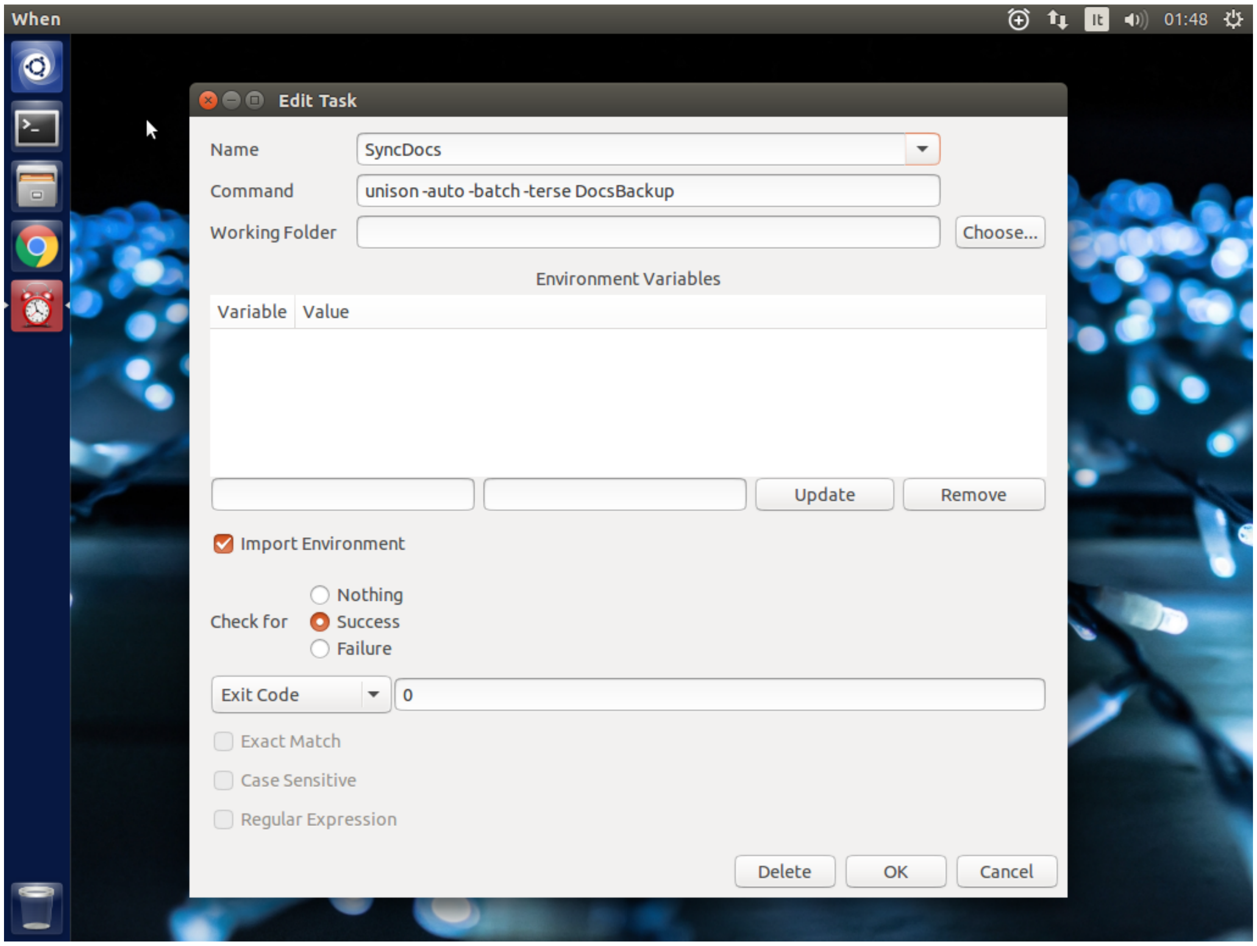
1. Give the profile a name, `DocsBackup` for brevity, and a meaningful description: something like "Backup documents to a remote location" would do the job.
2. Choose *Local* as synchronization kind because a NFS mounted file system appears as local to *Unison*.
3. Choose your main *Documents* folder as *first directory*, and browse to `/remote/<your_account>/backup/Documents` for the *second folder* using the *Other...* entry in the choice box.
4. Obviously leave the option for FAT partitions unchecked.
5. Click *Apply*.

A basic profile is now created, which is enough for our purposes. The profile name, `DocsBackup`, will be used in the command passed to **When** to let it perform the synchronization task.

To let **When** only do routine jobs, run the profile interactively by opening it (use the *Open* button after selecting the profile). A dialog box appears, to show that it's the first time that the folders are synchronized: accept it, and click *Go* in the *Unison* main window. Now *Unison* can be closed.

Create a Task

Click the **When** clock icon on the top panel, and select *Edit Tasks...* in the menu. The *Task* creation dialog box will open.



Name: SyncDocs
Command: unison -auto -batch -terse DocsBackup
Working Folder:

[✓] Import Environment

Check for: Nothing [✓ Success] Failure

Exit Code: 0

[ ] Exact Match
[ ] Case Sensitive
[ ] Regular Expression

As the screenshot suggest, a name has to be entered in the first dialog box field: we choose `SyncDocs`, which is mnemonic enough. In the *Command* entry, the following command line has to be entered:

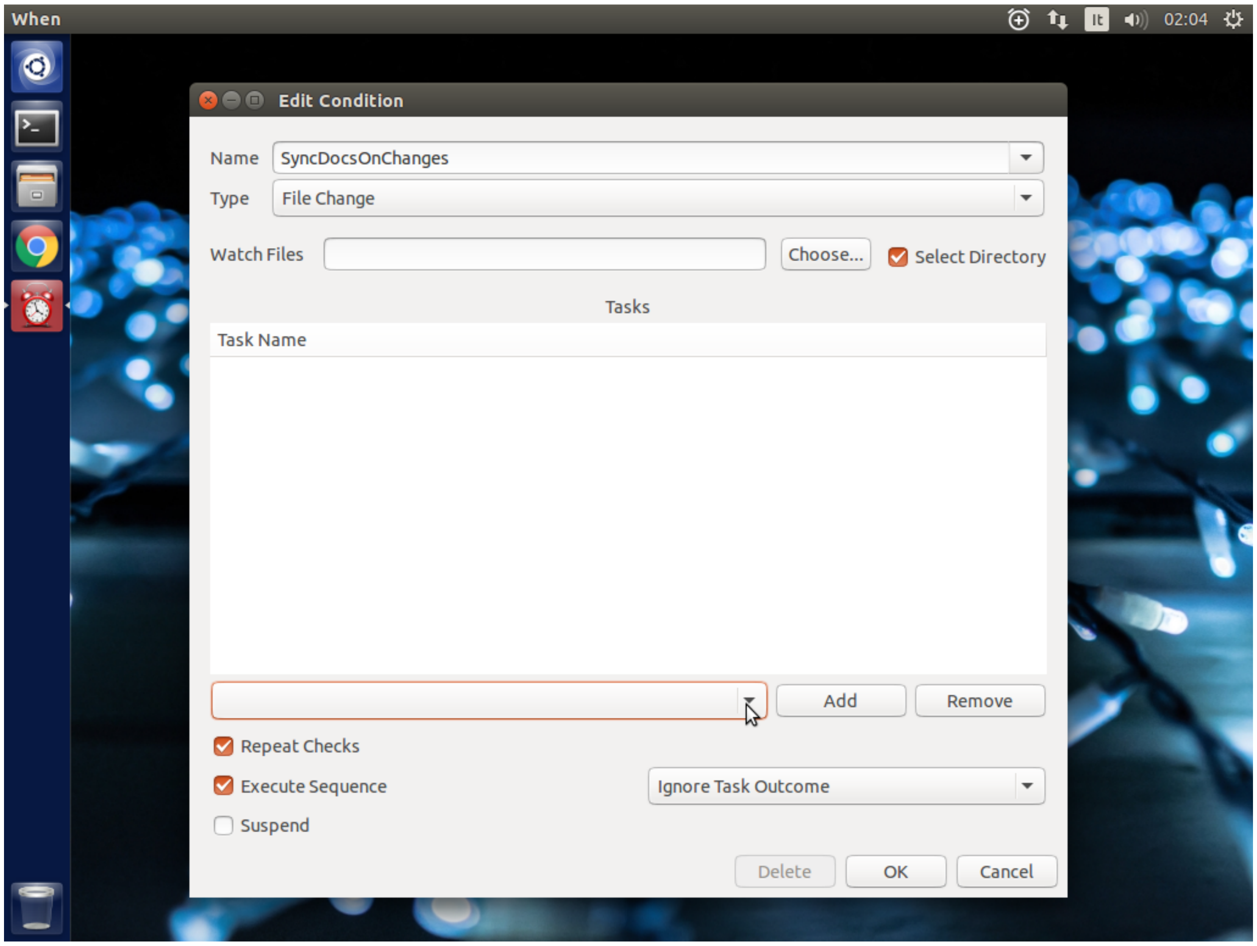
```
unison -auto -batch -terse DocsBackup
```

This tells *Unison* (the non-graphical utility) to perform a synchronization in automatic mode, asking no questions and with brief output. The other entries in the dialog box are left alone: the working directory is not influent, and we only care to know whether or not the synchronization task succeeded by interpreting the command exit status. As it mostly happens with command line utilities, *Unison* will return a zero exit code on success, and the other entries in the box just tell **When** to consider this.

Click *OK* to create the task.

Setup a Condition

We are interested in propagating changes in the source directory to the backup directory. The ideal solution is to create a condition based on *file and directory changes*. Click the **When** clock icon on the top panel, and select *Edit Conditions...* in the menu. The following dialog box will let us define such a condition.



Name: SyncDocsOnChanges
Type: File Change
Watch Files:
Choose...
[✓] Select Directory
Tasks
Task Name
Add
Remove
[✓] Repeat Checks
[✓] Execute Sequence
Ignore Task Outcome
[ ] Suspend
Delete
OK
Cancel

Then we will follow these steps:

1. Give the condition a meaningful name, such as `SyncDocsOnChanges`.
2. Select *File Change* in the drop-down list below.
3. Click the *Choose...* button and select the main documents folder (that is, `~/Documents`); alternatively the full path could be entered in the *Watch Files* field, which has the same effect.
4. Click the drop-down list under the list of tasks, and select `SyncDocs`, then click the *Add* button on its right.
5. Click *OK* to enter the new condition.

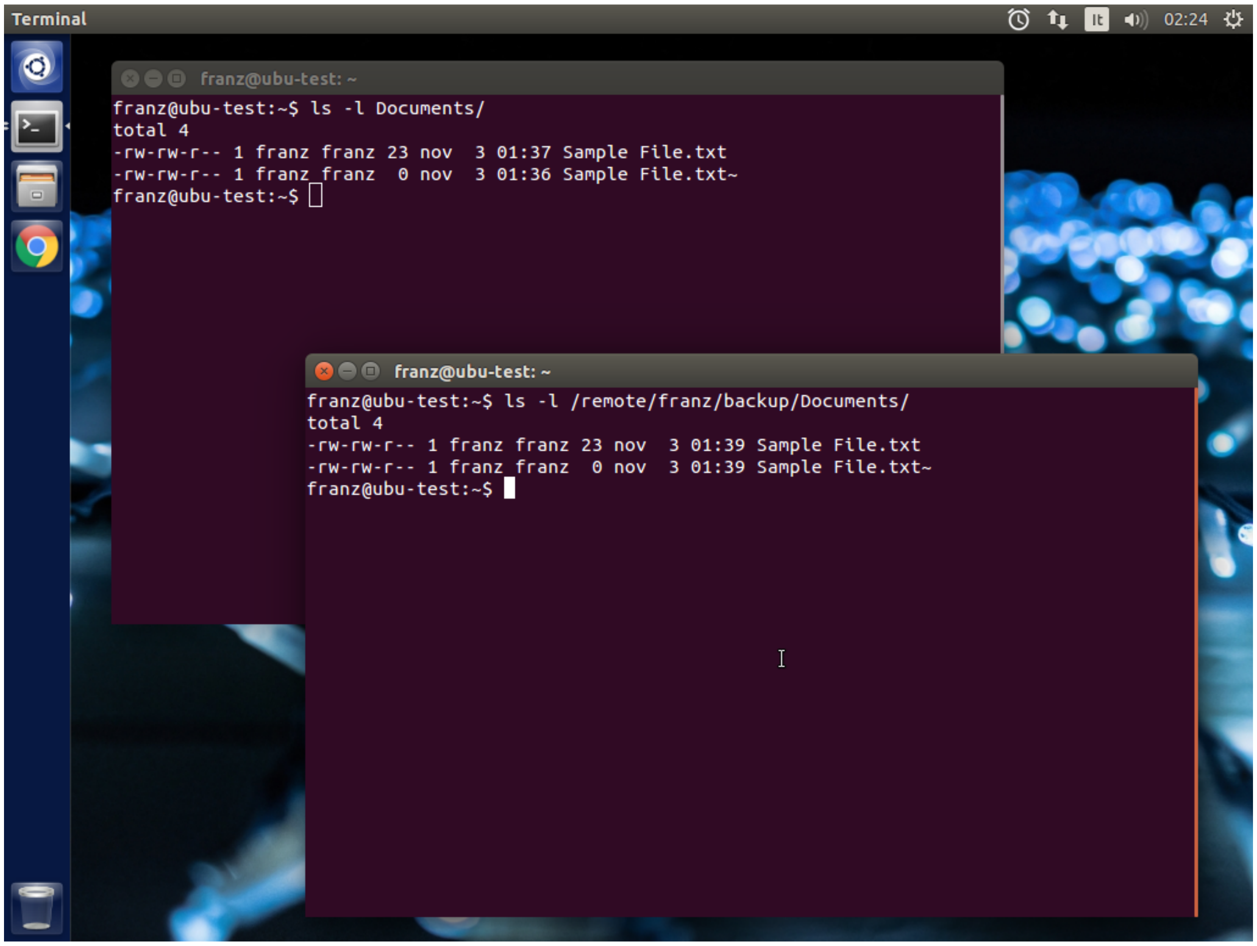
All the other fields should be left alone: in this way the checks are periodic (otherwise the synchronization would only take place once per session), while the other options are ininfluent in this case, as there is only one task for this condition.

Work and Let When do its Job

We are ready now: we should only check that changes in the source directory are reflected in the destination. A simple test will consist in the creation of a file in `~/Documents`:

```
$ cd
$ touch Documents/AnotherFile.txt
$ ls -l /remote/$USER/backup/Documents
```

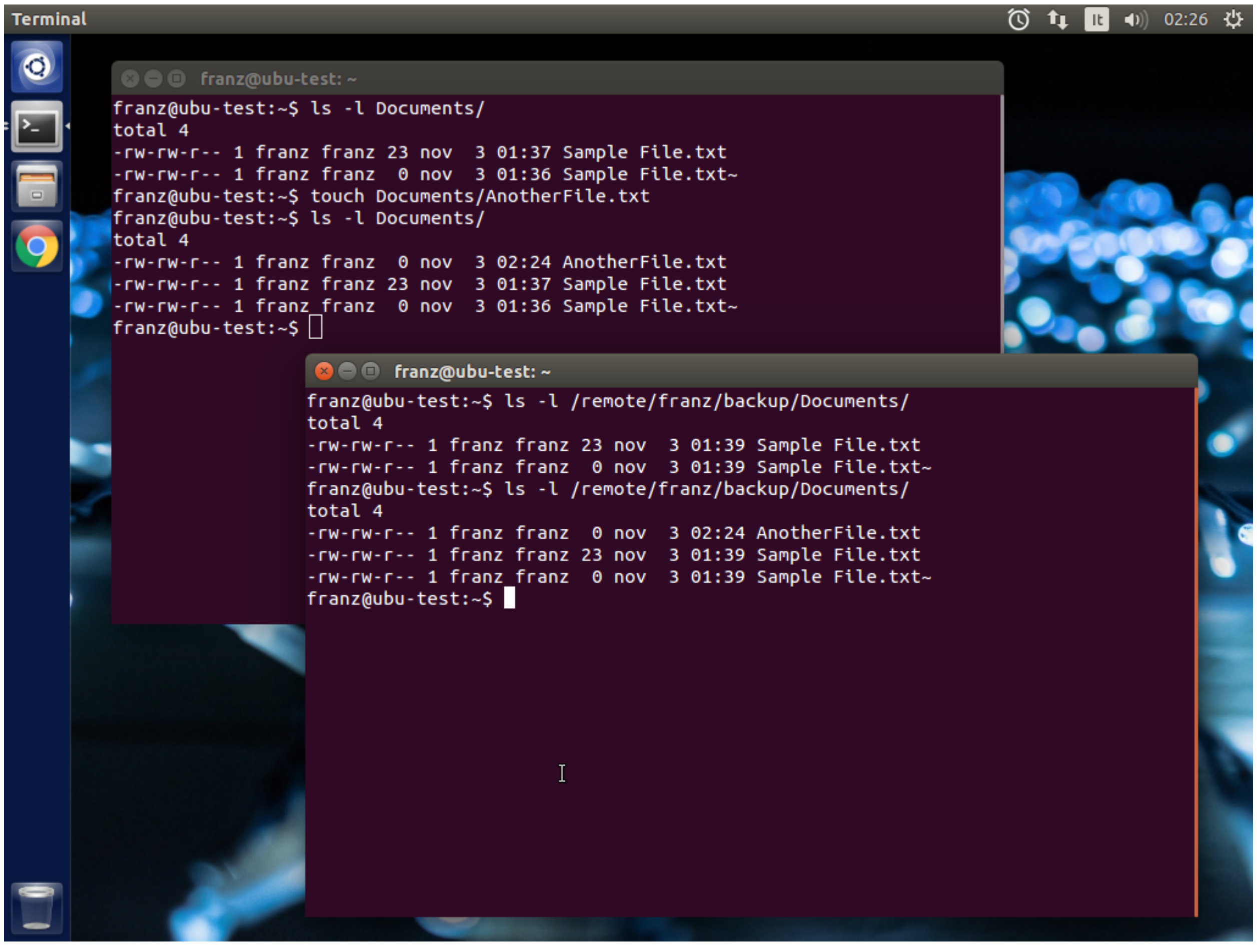
The following screenshots show how it worked: first is before creation



```
franz@ubu-test:~$ ls -l Documents/
total 4
-rw-rw-r-- 1 franz franz 23 nov  3 01:37 Sample File.txt
-rw-rw-r-- 1 franz franz  0 nov  3 01:36 Sample File.txt~
franz@ubu-test:~$
```

```
franz@ubu-test:~$ ls -l /remote/franz/backup/Documents/
total 4
-rw-rw-r-- 1 franz franz 23 nov  3 01:39 Sample File.txt
-rw-rw-r-- 1 franz franz  0 nov  3 01:39 Sample File.txt~
franz@ubu-test:~$
```

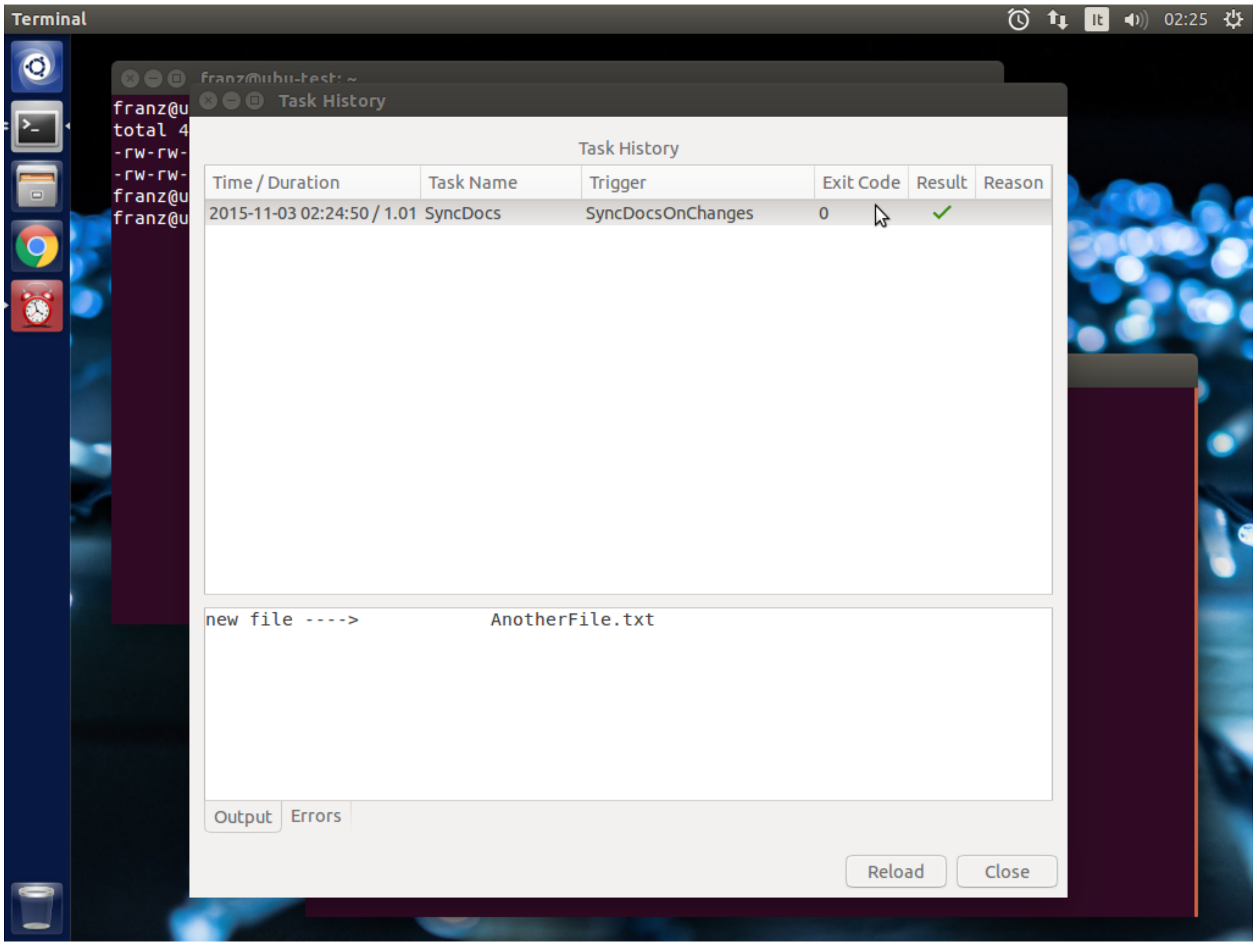
and after:



```
franz@ubu-test:~$ ls -l Documents/
total 4
-rw-rw-r-- 1 franz franz 23 nov  3 01:37 Sample File.txt
-rw-rw-r-- 1 franz franz  0 nov  3 01:36 Sample File.txt~
franz@ubu-test:~$ touch Documents/AnotherFile.txt
franz@ubu-test:~$ ls -l Documents/
total 4
-rw-rw-r-- 1 franz franz  0 nov  3 02:24 AnotherFile.txt
-rw-rw-r-- 1 franz franz 23 nov  3 01:37 Sample File.txt
-rw-rw-r-- 1 franz franz  0 nov  3 01:36 Sample File.txt~
franz@ubu-test:~$
```

```
franz@ubu-test:~$ ls -l /remote/franz/backup/Documents/
total 4
-rw-rw-r-- 1 franz franz 23 nov  3 01:39 Sample File.txt
-rw-rw-r-- 1 franz franz  0 nov  3 01:39 Sample File.txt~
franz@ubu-test:~$ ls -l /remote/franz/backup/Documents/
total 4
-rw-rw-r-- 1 franz franz  0 nov  3 02:24 AnotherFile.txt
-rw-rw-r-- 1 franz franz 23 nov  3 01:39 Sample File.txt
-rw-rw-r-- 1 franz franz  0 nov  3 01:39 Sample File.txt~
franz@ubu-test:~$
```

To check outcome directly from the **When** interface, we can open the history window, by choosing *History...* in the applet menu.



Task History

Time / Duration	Task Name	Trigger	Exit Code	Result	Reason
2015-11-03 02:24:50 / 1.01	SyncDocs	SyncDocsOnChanges	0	✓	

```
new file ---->          AnotherFile.txt
```

This dialog box also shows the (brief) output of the command, which is useful to identify task outcomes. If we click on the list items, the panes below will show output (*stdout* and *stderr*) for the selected task.

Housekeeping

In this tutorial we will instruct **When** to perform some simple housekeeping in the *Documents* directory when the session has been idle for a while. For the example we will use a minimal shell script that removes the files that end in the *tilde* character (usually backups) and sends them to the trash can. We need to use the `trash` command, which can be installed with the *trash-cli* package:

```
$ sudo apt-get install trash-cli
```

An alternative could be to directly remove the files, but this would be more dangerous and we want to keep some kind of control on what is actually removed from the disk.

Write the Shell Script

Our script is essential, as said above, but nothing forbids to let it do more complex tasks. To keep the things somewhat standard, we will put the script in the `~/.local/bin` directory. At a terminal prompt, do the following:

```
$ mkdir ~/.local/bin
$ cd ~/.local/bin
$ gedit housekeep.sh
```

When the Gnome editor starts, enter the following text:

```
#!/bin/sh
find . -path ./.local/share/Trash -prune \
    -o -type f -name '*~' \
    -exec echo '{}' \; \
    -exec trash -f '{}' \;
```

save the file, exit the editor and from the same terminal window run

```
$ chmod a+x housekeep.sh
```

to make the script executable. The reason for the line that discards stuff in `./.local/share/Trash` is that we don't want files already in the trash bin to be handled again, and we are pretty sure that such a folder only exists in the user home directory – so that this limitation only has effect when the startup directory is the home directory.

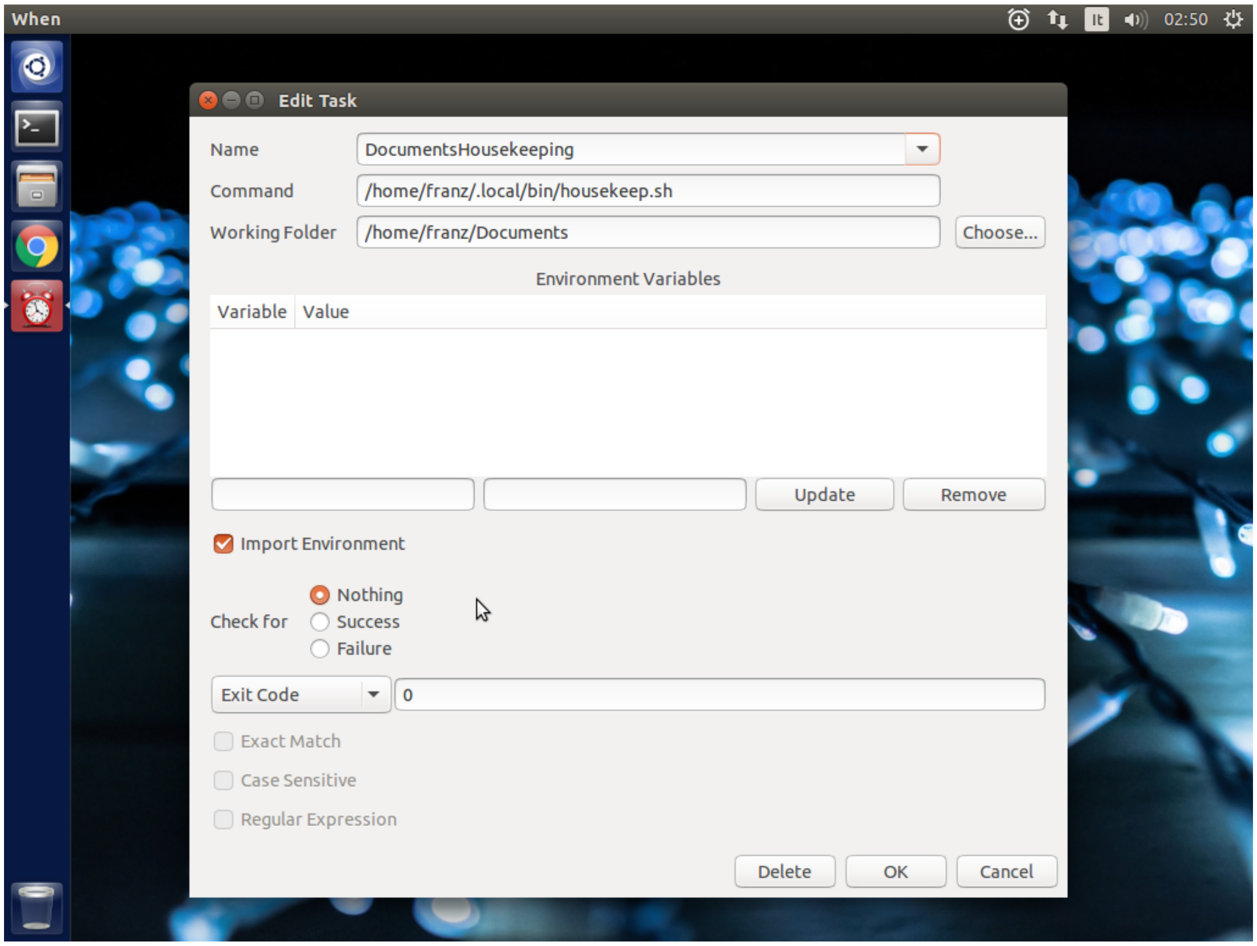
The `housekeep.sh` script is available here.

Create the Task

From the **When** menu select the *Edit Tasks...* entry. When the *task editor* box shows up, choose a meaningful name for the task: `DocumentsHousekeeping` will do the job. Then insert the following text in the *Command* field:

```
/home/<your_account>/.local/bin/housekeep.sh
```

(where `<your_account>` should be changed to your account name). Hit the *Choose...* button to select the working folder and navigate to select the `~/Documents` directory. This is actually the reason why we just told the `find` command to start in the current directory: **When** will change directory for us before starting the script, and we can use the same script to create tasks that perform housekeeping in other directories, just changing the startup directory.



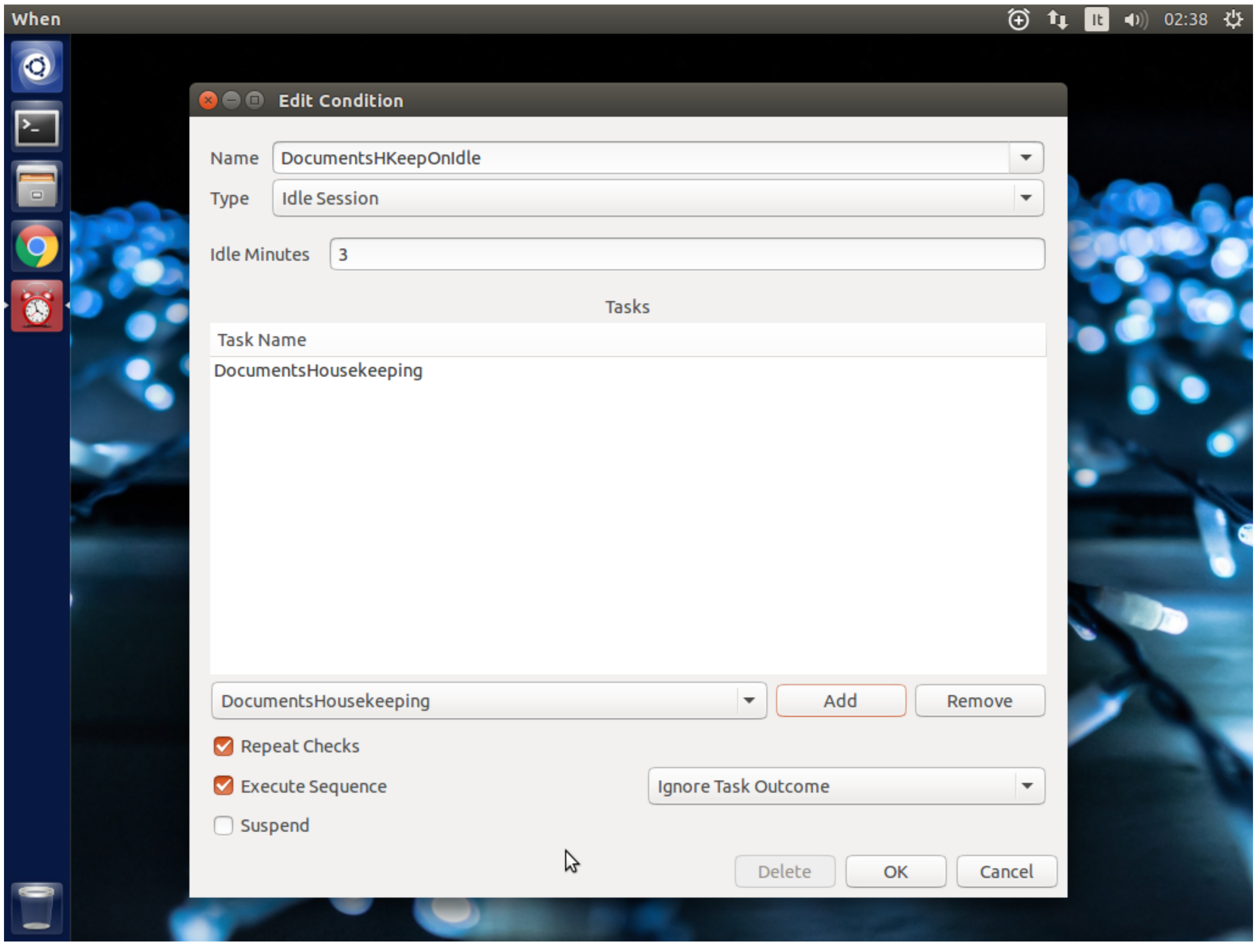
Name: DocumentsHousekeeping
Command: /home/franz/.local/bin/housekeep.sh
Working Folder: /home/franz/Documents
[✓] Import Environment
Check for: [✓ Nothing] Success Failure
Exit Code: 0
[ ] Exact Match
[ ] Case Sensitive
[ ] Regular Expression

Since we really don't care about task outcome and we don't want **When** to throw an error when this task fails, we also select to check for *Nothing* as outcome.

Click *OK* to accept the task.

Setup the Condition

We want this task to occur whenever the session has been idle for, say, three minutes. It's not a very expensive task, so we accept it to run more than once per session. To create the condition, select *Edit Conditions...* from the applet menu. In the *condition editor* choose a meaningful name for the item, such as `DocumentsHKeepOnIdle`, and choose *Idle Session* from the drop-down list. Specify *3* in the *Idle Minutes* field, then using the drop-down list below the task list, choose the `DocumentsHousekeeping` task and click the *Add* button on the right. We can leave the other entries alone.



Name: DocumentsHKeepOnIdle
Type: Idle Session
Idle Minutes: 3

Tasks

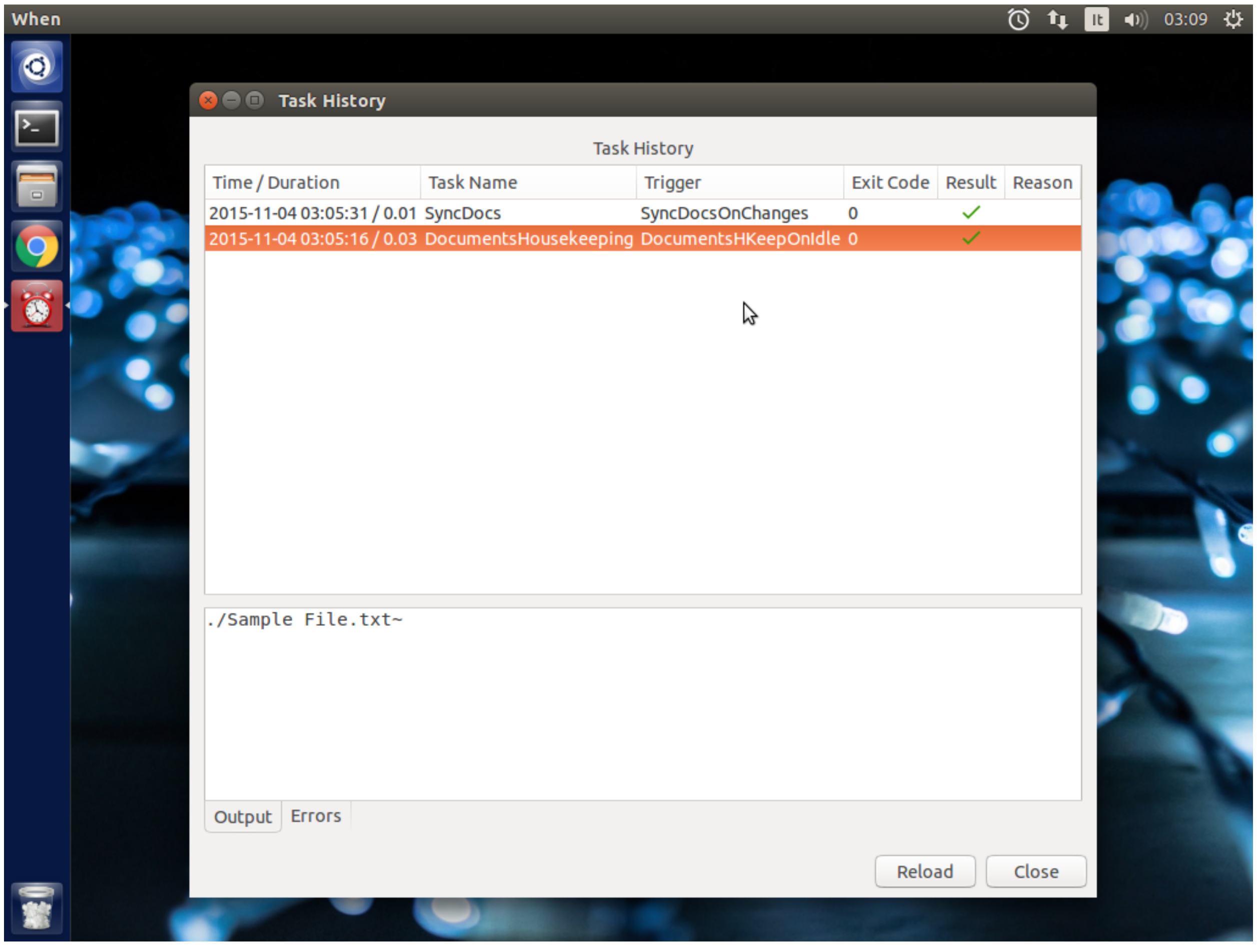
Task Name
DocumentsHousekeeping

- [x] Repeat Checks
- [x] Execute Sequence — Ignore Task Outcome
- [ ] Suspend

Click *OK* to accept the condition, and we're done.

Verify that Everything Worked

After three or four (**When** is a lazy applet, though) minutes, you can open the *History* box by selecting *Task History...* from the applet menu. The window will show `DocumentsHousekeeping` triggered by `DocumentsHKeepOnIdle` in the main list, possibly among other tasks.



Task History

Time / Duration	Task Name	Trigger	Exit Code	Result	Reason
2015-11-04 03:05:31 / 0.01	SyncDocs	SyncDocsOnChanges	0	✓	
2015-11-04 03:05:16 / 0.03	DocumentsHousekeeping	DocumentsHKeepOnIdle	0	✓	

```
./Sample File.txt~
```

If you click the task line, you can verify what happened in the *Output* and *Errors* tab below: because the script writes the name of each file it deletes to *stdout*, the file names appear in the *Output* pane. Also note that the desktop trash bin is now full, because `Sample File.txt~` was moved there. As the condition from the previous example (`SyncDocsOnChanges`) has not been removed, it has been triggered by the above defined task some seconds later.

Automatic Import

This example shows how to automatically import files from an external storage device, such as an USB stick, when it's automatically mounted by the desktop manager. Suppose we're using an USB stick to gather data and move it from some device to our workstation. We assume that the USB stick has been given a label (we'll call it `USB2GB` in this example) and that the device always writes to the same `Data` directory on the stick, with no subdirectory: this makes things easier, because we can use `cp` or `mv` to transfer files to the hard disk.

Ubuntu always mounts external storage devices under the `/media` directory, using `/media/<label>` as the actual mount point. So we can presume that our USB stick will be mounted on `/media/<your_account>/USB2GB`, and we can also be reasonably sure that, if there is a `/media/<your_account>/USB2GB/Data` directory around just after insertion of a storage device, it must be the place to gather data from. Naturally there should be better ways to determine this. What we will do is blindly copy all files found in the device's `Data` folder to `~/Documents/Gathered`, which has been created for this purpose using the following command:

```
$ mkdir ~/Documents/Gathered
```

A shell script will be used to perform the task, just because we'd like to:

- be notified by a badge reporting the operation outcome
- avoid to clutter the *Command* entry in the **When** *task editor* box
- have the script to unmount the USB stick if the copy succeeds.

The last step is less likely to be needed in the real world, as this would cancel any possibility to read the contents of the device unless turning off **When**.

We will make use of an USB stick (labeled `USB2GB` through *parted* or *GParted* or any other disk labeling utility) which has a `Data` directory with some crafted CSV files (ending in `.csv`).

Create the Shell Script

Using the same technique shown in the second example, we will create a script called `gather_data.sh` in `~/.local/bin`, containing the following text:

```
#!/bin/bash

# this script expects two variables to be defined:
# DEVICE_LABEL is the label given to the removable storage device
# DESTINATION is the destination folder

if [ -z "$DEVICE_LABEL"]; then exit 2; fi
if [ -z "$DESTINATION"]; then exit 2; fi

# shortcuts
SOURCE_BASE="/media/$USER/$DEVICE_LABEL"
SOURCE=$SOURCE_BASE/Data

# exit if it's not the right USB key
if [ ! -d "$SOURCE" ]; then
    exit 2
fi

# copy data from source base to destination
cp -f $SOURCE/*.csv $DESTINATION

# if the task was successful show a badge, if not When enters an error state
if [ "$?" = "0" ]; then
  gvfs-mount -u $SOURCE_BASE
  notify-send -i info "Data Gatherer" "Files successfully transferred, remove device"
else
  exit 2
fi
```

Once written, do a `chmod a+x gather_data.sh` in the same directory from a terminal window.

The `gather_data.sh` script is available at this location.

Note: *A Tip for Photographers*

Digital cameras nowadays use mostly SD cards (which contain well known directories, such as `DCIM`) to store pictures: with adequate changes (such as copying `*.jpg` files from a different directory) this script can be helpful to transfer photos whenever a SD card is inserted. You can also use it for storage devices different from SD cards, as long as you correctly name the default dynamic mount point.

Create the Task

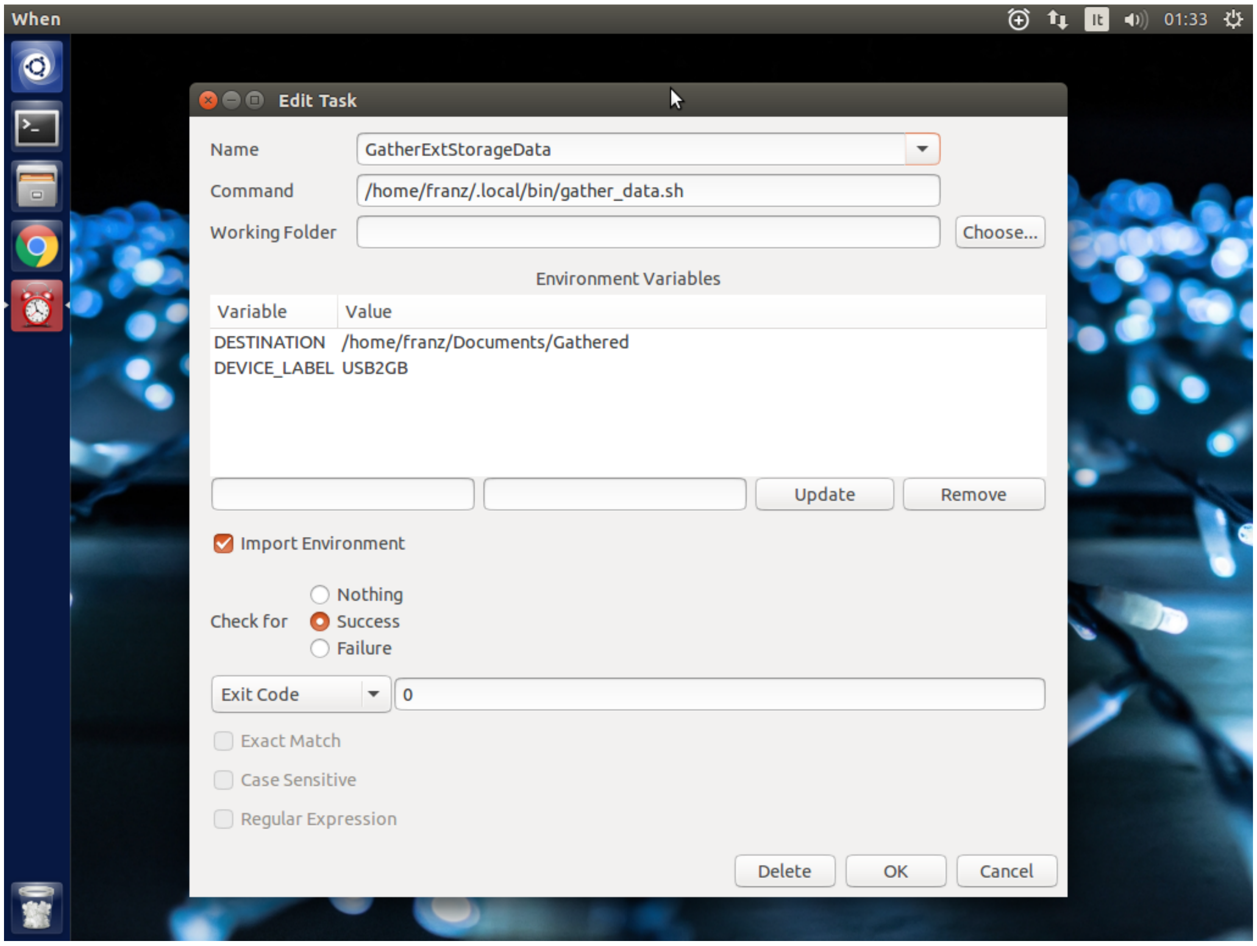
The corresponding *task* will need to consider the exit status of our script, since we rely on it to show a failure badge on task failure: this is the default with **When** when it's not instructed to avoid notifications on task failures. So we will create a task with an adequate name (`GatherExtStorageData`) that

- defines the two needed variables: `DEVICE_LABEL` and `DESTINATION` (respectively with the label given to the external storage device and the destination folder)
- checks that the script exit code is `0`.

To do this, we must open the *task editor* window by selecting *Edit Tasks...* from the applet menu and then follow these steps:

1. enter `GatherExtStorageData` in the *Name* field
2. enter `/home/<your_account>/.local/bin/gather_data.sh` in the *Command* field (`<your_account>` has to be replaced by your account name)
3. define the two needed variables, by writing the variable name (*remember that names are case sensitive!*) in the entry below the variable list, and its value in the adjacent text field and then hitting the *Update* button: `DEVICE_LABEL` should contain `USB2GB`, and `DESTINATION` the full path to the destination directory, that is `/home/<your_account>/Documents/Gathered` where `<your_account>` is replaced by your account name.

The other entries must be left alone: the default *task* definition box is already set up to look for task success by checking that the exit code is zero. This is how the *task editor* box looks like after we defined the task:



Name: GatherExtStorageData
Command: /home/franz/.local/bin/gather_data.sh
Working Folder:

Variable	Value
DESTINATION	/home/franz/Documents/Gathered
DEVICE_LABEL	USB2GB

[✓] Import Environment

Check for: Nothing [✓ Success] Failure

Exit Code: 0

☐ Exact Match
☐ Case Sensitive
☐ Regular Expression

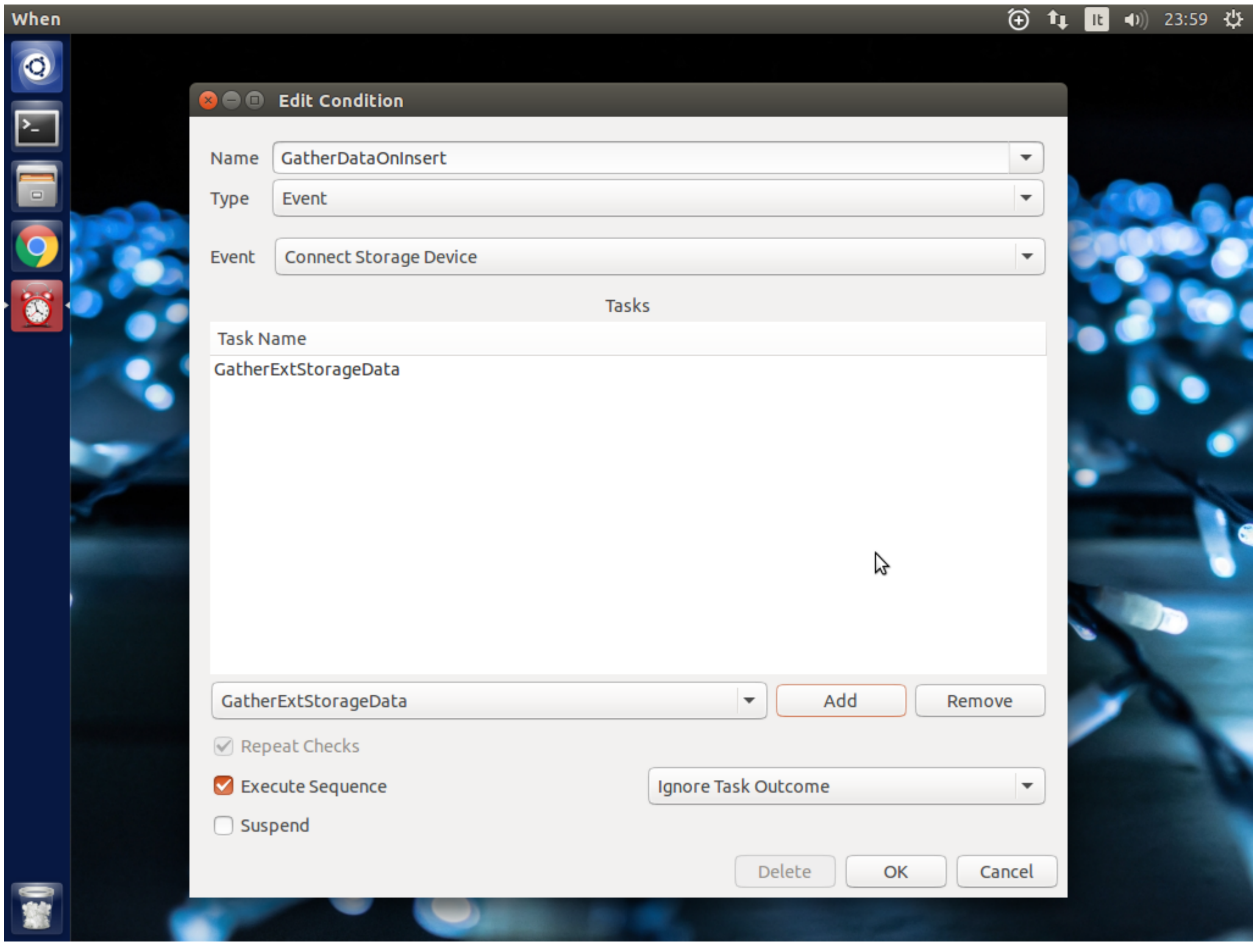
You can click the *OK* button to store the *task item*.

Setup the Condition

The most adequate condition type here is the *event* based one: it allows to choose a subtype that causes it to occur on storage device connection. **When** is quite generic in this case, and does not actually communicate to the user any details about the actual storage device. However, knowing the expected dynamic mount point helps in writing scripts – like the one above – that only work when the correct device has been inserted.

To define the condition we will select *Edit Conditions...* entry from the applet menu and carry the following operations when the dialog box appears:

1. give the condition a meaningful name, such as `GatherDataOnInsert`
2. select *Event* from the *Type* drop-down box: the dialog layout will change and a second drop-down list appears
3. select *Connect Storage Device* from the following drop-down list, that just appeared
4. using the combo box under the task list, choose `GatherExtStorageData` and hit the *Add* button.



Name: GatherDataOnInsert
Type: Event
Event: Connect Storage Device

Task Name
GatherExtStorageData

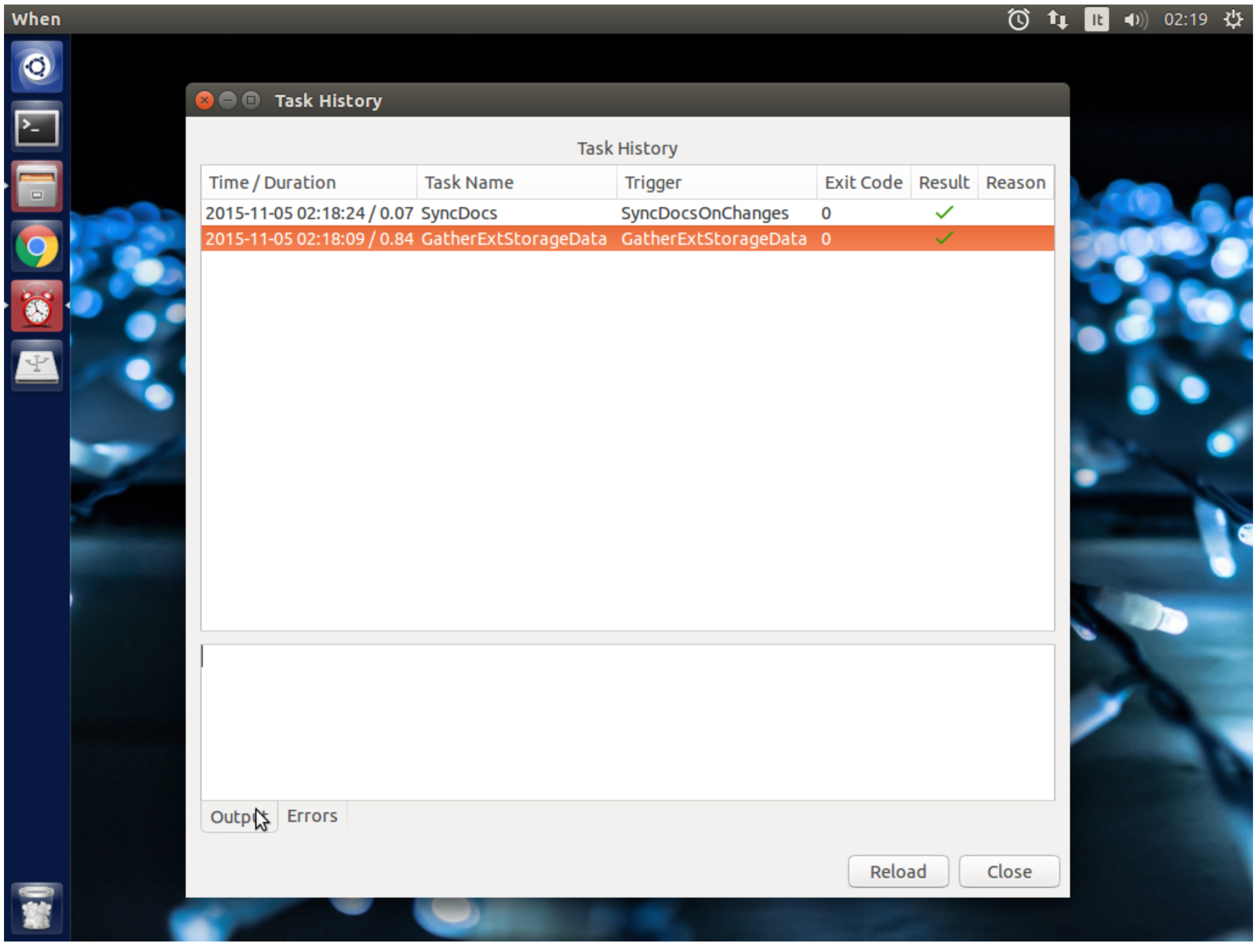
GatherExtStorageData

- [✓] Repeat Checks
- [✓] Execute Sequence — Ignore Task Outcome
- [ ] Suspend

This is sufficient and other entries could be left alone. Click *OK* to accept.

Cause Something to Happen

We just have to insert the USB key to let **When** work now. After a while a notification will inform us that the files have been successfully copied to the destination directory. If we open the *task history* box by choosing *Task history...* from the applet menu



Task History

Time / Duration	Task Name	Trigger	Exit Code	Result	Reason
2015-11-05 02:18:24 / 0.07	SyncDocs	SyncDocsOnChanges	0	✓	
2015-11-05 02:18:09 / 0.84	GatherExtStorageData	GatherExtStorageData	0	✓	

we can check that the task actually succeeded at any time. Using `ls`, for instance, or *Nautilus*, you can also verify that all files have been copied to the destination.

The Tutorial is an Ongoing Task

This tutorial was formerly a project by itself, and now is part of a bigger project focusing on structured documentation, also to allow it to grow with time: for now it consists of very simple examples, but **When** can be also used to automate complex tasks. It can be useful for developers, photographers, people that need to automate data gathering or processing and so on: feel free to provide or just even suggest more examples using the *issue* mechanism in the documentation repository.

CHAPTER 7

Credits and Resources

Open Source Software relies on collaboration, and I'm more than happy to receive help from other developers. Here I'll list the main contributions.

- Adolfo Jayme-Barrientos, aka fitojb for the Spanish translation

Also, I'd like to thank everyone who contributes to the development of **When** by commenting, filing bugs, suggesting features and testing. Every kind of help is welcome.

The top panel icons and the emblems used in the application were selected within Google's Material Design icon collection.

The application icon has been created by Rafi at GraphicsFuel.

Resources

As said above, this software is designed to run mainly on Ubuntu and the chosen framework is *Python 3.x* with *PyGObject* (*GTK 3.0*); the interface is developed using the *Glade* interface designer. The resources I found useful are:

- Python 3.x Documentation
- PyGTK 3.x Tutorial
- PyGTK 2.x Documentation
- PyGObject Documentation
- GTK 3.0 Documentation
- DBus Documentation
- pyinotify Documentation

The guidelines specified in UnityLaunchersAndDesktopFiles have been roughly followed to create the launcher from within the application.

Many hints and valuable information have been found on StackOverflow and the other sites in the StackExchange network.

Bugs and Errors

When is hosted on GitHub: the repository contains the most recent stable code as well as developement and feature branches. The *master* branch might include more recent code with respect to the packaged distributions. The repository for **When** also gives access to the bug tracking system, in the form of the *Issues* mechanism. *Issues* can be used to provide information on bugs or features that could make **When** more useful.

Before filing an *issue* please consider that

- in the case of a bug some data are needed:
  - **When** version
  - Linux distribution and complete version
  - Python 3.x detailed version
  - How **When** was installed (which package, or how source was obtained)
  - Steps to reproduce the problem.

  Before filing a bug please verify that there is no open equivalent issue, or that the issue is not a particular case of an already open one.
- for a feature request the following should be taken into account:
  - whether or not it would make the applet more useful or usable
  - if the feature being requested is just a shortcut for something that can already be done via configuration (for instance, adding an event that could be provided using a *signal handler*)
  - how it would impact on **When** in terms of weight and responsiveness
  - the impact that it would have on backward compatibility.

  Consider that **When** should try to remain as small as possible, it already eats up around 20MBytes as it is: most effort in its development should go towards simplification and extendability via external tools.

The repository is also the starting point for other forms of contributions. There is a separate documentation for contributors, that tries to cover most possible areas.

CHAPTER 8

License (BSD)

Copyright (c) 2015-2016, Francesco Garosi

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of when-command nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.