Astrality Documentation

Release 1.1.0

Jakob Gerhard Martinussen

Nov 19, 2019

Documentation

CHAPTER 1

Table of Contents

1.1  Astrality - A Dynamic Configuration File Manager

TL;DR: Automatically deploy dotfiles. Grouped into modules with dynamic behaviour.

1.1.1 What does it do?

Astrality is a flexible tool for managing configuration files, inspired by GNU Stow and Ansible.

Let's begin with a list of some of Astrality's key features:

- Manage and deploy configuration files according to a central YAML config file.
- Group related configuration into *modules*.
- Conditionally enable modules based on environment variables, OS, installed programs and shell commands.
- Copy and/or symlink files.
- Execute shell commands.
- Compile Jinja2 templates templates to target destinations.
- Dynamically manipulate context values used during jinja2 compilation.
- Automatically re-deploy dotfiles when source content is modified.
- Subscribe to pre-defined events, such as local daylight, and execute actions accordingly.
- Fetch modules from GitHub.
- Restore files created and/or overwritten by modules.

Take a look at the tutorial for managing a dotfile repository, or see the full documentation for all available functionality. Feel free to drop by our Gitter room when getting started.

Here is gif demonstrating how Astrality is used to:

1) Automatically change the desktop wallpaper based on the sun's position in the sky.
2) Dynamically change the font size, and implicitly the bar height, of polybar.
3) Simultaneously change the color scheme of alacritty, kitty, and polybar at the same time.

1.1.2 Getting started

Prerequisites

Astrality requires python 3.6 or greater. Check your version by running `python --version`.

Installation

`astrality-git` is published on the AUR for ArchLinux users. Otherwise, you can install Astrality using `pip`:

Create a new virtualenv for python 3.6 (or use your system python 3.6 if you prefer). Install Astrality from PyPI like so:

```
$ python3.6 -m pip install astrality
```

You should now be able to start *astrality* from your command line, but first, let us create an example configuration:

```
$ astrality --create-example-config
```

Take a look at the generated example configuration at `~/.config/astrality`. Now start *astrality*:

```
$ astrality
```

Configuration and further documentation

I recommend taking a look at the full documentation of Astrality hosted at Read the Docs.

1.2 Configuration

1.2.1 The Astrality configuration directory

The configuration directory for astrality is determined in the following way:

- If `$ASTRALITY_CONFIG_HOME` is set, use that path as the configuration directory, else...
- If `$XDG_CONFIG_HOME` is set, use `$XDG_CONFIG_HOME/astrality`, otherwise...
- Use `~/.config/astrality`.

The resulting directory path can be displayed by running:

```
$ astrality --help
usage: Astrality ...
...
The location of Astralitys configuration directory is:
"/home/jakobgm/.dotfiles/config/astrality".
...
```

This directory path will be referred to as `$ASTRALITY_CONFIG_HOME` in the rest of the documentation.

1.2.2 The Astrality configuration files

There are three configuration files of importance in `$ASTRALITY_CONFIG_HOME`:

`astrality.yml` Global configuration options for Astrality.

`modules.yml` Here you define modules you want to use.

`context.yml` Context values used for placeholder substitution in compiled templates.

If `$ASTRALITY_CONFIG_HOME/astrality.yml` does not exist, an example configuration directory will be used instead.

You can also copy over this example configuration directory as a starting point for your configuration by running:

```
$ astrality --create-example-config
Copying over example config directory to "/home/example_username/.config/astrality".
```

You should now edit `astrality.yml`, `modules.yml`, and `context.yml` to fit your needs.

1.2.3 The configuration file syntax

Astrality's configuration files uses the `YAML` format. The syntax should be relatively self-explanatory when looking at the example configuration. If you still want a basic overview, take a look at the Ansible YAML syntax documentation for a quick primer.

Command substitution in configuration files

Astrality's configuration files are themselves templates that are compiled and interpreted at startup. Using templating features in astrality configuration files is usually unnecessary.

But sometimes it is useful to insert the result of a shell command within a configuration file, such as "context.yml". You can use command substitutions in order to achieve this:

- **Command substitution:** `{{ 'some_shell_command' | shell }}` is replaced with the standard output resulting from running `some_shell_command` in a `bash` shell.

  You can set a timeout and/or fallback value for command substitutions. See the *documentation* for the shell filter.

Note: Shell commands are always executed from the same directory as the file which contains the command substitution.

If you need to refer to paths outside this directory, you can use absolute paths, e.g. `{{ 'cat ~/.home_directory_file' | shell }}`.

1.2.4 Astrality configuration options

Global Astrality configuration options are specified in `astrality.yml` within a dictionary named `astrality`, i.e.:

```
# Source file: $ASTRALITY_CONFIG_HOME/astrality.yml
astrality:
    hot_reload_config: true
    startup_delay: 10
```

Avalable configuration options:

`hot_reload_config`: *Default:* `false`

If enabled, Astrality will watch for modifications to `astrality.yml`, `modules.yml`, and `context.yml`.

When one of these are modified, Astrality will perform all *exit actions* in the old configuration, and then all *startup actions* from the new configuration.

Ironically requires restart if enabled.

Useful for quick feedback when editing your configuration.

`startup_delay`: *Default:* `0`

Delay Astrality on startup, given in seconds.

Useful when you depend on other startup scripts before Astrality startup, such as reordering displays.

1.2.5 Where to go from here

What you should read of the documentation from here on depends on what you intend to solve by using Astrality. The most central concepts are:

- *Templating* explains how to write configuration file templates.
- *Modules* specify which templates to compile, when to compile them, and which commands to run after they have been compiled.
- *Event listeners* define types of events which modules can listen to and change their behaviour accordingly.

These concepts are relatively interdependent, and each documentation section assumes knowledge of concepts explained in earlier sections. If this is the first time you are reading this documentation, you should probably just continue reading the documentation in chronological order.

1.3 Templating

1.3.1 Template files

Templates can be of any file type, named whatever you want, and placed at any desirable path. If you want to write a template for a file named "example.conf" it is recommended that you name it "template.example.conf".

1.3.2 Context

When you write templates, you use `{{ placeholders }}` which Astrality replaces with values defined in so-called `context` sections defined in `$ASTRALITY_CONFIG_HOME/context.yml`.

Here is an example which defines context values in "context.yml":

```
# $ASTRALITY_CONFIG_HOME/context.yml

machine:
    user: jakobgm
    os: linux
    hostname: hyperion

fonts:
    1: FuraCode Nerd Font
    2: FuraMono Nerd Font
```

Warning: Context keys (anything left of a colon) can only consist of ASCII letters, numbers and underscores. **No spaces are allowed**.

Inserting context variables into your templates

You should now be able to insert context values into your templates. You can refer to context variables in your templates by using the syntax `{{ context_section.variable_name }}`.

Using the contexts defined above, you could write the following template:

```
font-type = '{{ fonts.1 }}'
home-directory = /home/{{ machine.user }}
machine-name = {{ machine.hostname }}
```

When Astrality *compiles your template* the result would be:

```
font-type = 'FuraCode Nerd Font'
home-directory = /home/jakobgm
machine-name = hyperion
```

Hint: You can create arbitrarily nested structures within context sections. For instance:

```
cosmetics:
    fonts:
        1:
            family: FuraCode
            font_size: 13
        2:
            family: FuraMono
            font_size: 9
```

And refer to those nested variables with "dotted" syntax `{{ cosmetics.fonts.1.family }}`.

The `env` context

Astrality automatically inserts a context section at runtime named `env`. It contains all your environment variables. You can therefore insert environment variables into your templates by writing:

```
{{ env.ENVIRONMENT_VARIABLE_NAME }}
```

Undefined context values

When you refer to a context value which is not defined, it will be replaced with an empty string, and logged as a warning in Astrality's standard output.

Default fallback context values

Sometimes you want to refer to context variables in your templates, but you want to insert a fallback value in case the context variable is not defined at compile time. This is often the case when referring to environment variables. Defining a fallback value is easy:

```
{{ env.ENVIRONMENT_VARIABLE_NAME or 'defualt value' }}
```

Integer placeholder resolution

There exists another way to define fallback values, which sometimes is much more useful.

Let's define context values with integer names:

```
# $ASTRALITY_CONFIG_HOME/context.yml

fonts:
    1: FuraCode Nerd Font
    2: FuraMono Nerd Font
```

You can now write the following template:

```
primary-font = '{{ fonts.1 }}'
secondary-font = '{{ fonts.2 }}'
tertiary-font = '{{ fonts.3 }}'
```

And it will be compiled to:

```
primary-font = 'FuraCode Nerd Font'
secondary-font = 'FuraMono Nerd Font'
tertiary-font = 'FuraMono Nerd Font'
```

With other words, references to *non-existent* numeric context identifiers are replaced with the greatest *available* numeric context identifier at the same indentation level.

Hint: This construct can be very useful when you are expecting to change the underlying context of templates. Defining font types and color schemes using numeric identifiers allows you to switch between themes which define a different number of fonts and colors to be used!

1.3.3 Advanced templating

Astrality templating uses `Jinja2` under the hood. If you want to apply more advanced templating techniques than the ones described here, you can use the extended templating features available in the Jinja2 templating engine. Visit Jinja2's templating documentation for more information.

Useful constructs include:

Filters: For manipulating context variables before insertion.

Template inheritance: For reuse of templates with common sections.

Iterating over context values: For using both the context *name* and *value* in configuration files.

Conditionals: For only including template content if some conditions(s) are satisfied.

The `shell` filter

Astrality provides an additional `shell` template filter in addition to the standard Jinja2 filters. The syntax is:

```
{{ 'shell command' | shell }}
```

Note: Shell commands are run from the directory which contains the configuration for the template compilation, most often `$ASTRALITY_CONFIG_HOME`. If you need to refer to paths outside this directory, you can use absolute paths, e.g. `{{ 'cat ~/.bashrc' | shell }}`.

You can specify a timeout for the shell command given in seconds:

```
{{ 'shell command' | shell(5) }}
```

The default timeout is 2 seconds.

To provide a fallback value for functions that time out or return non-zero exit codes, do:

```
{{ 'shell command' | shell(1.5, 'fallback value') }}
```

Caution: The quotes around the shell command are important, since if you ommit the quotes, you end up refering to a context value instead. Though, this *can* be done intentionally when you have defined a shell command in a context variable.

1.3.4 How to compile templates

Now that you know how to write Astrality templates, you might wonder how to actually *compile* these templates. You can instruct Astrality to compile templates by defining a module in "$ASTRALITY_CONFIG_HOME/modules.yml". More on this on the next page of this documentation, but here is a simple example:

Let us assume that you have written the following template:

```
# Source: $ASTRALITY_CONFIG_HOME/templates/some_template

current_user={{ host.user }}
```

Where you want to replace `{{ host.user }}` with your username. Let us define the context value used for insertion in "$ASTRALITY_CONFIG_HOME/context.yml":

```
# Source: $ASTRALITY_CONFIG_HOME/context.yml

host:
    user: {{ env.USER }}
```

In order to compile this template to `$XDG_CONFIG_HOME/config.ini` we write the following module, which will compile the template on Astrality startup:

```
# Source: $ASTRALITY_CONFIG_HOME/modules.yml

my_module:
    compile:
        - content: templates/template
          target: $XDG_CONFIG_HOME/config.ini
```

Now we can compile the template by starting Astrality:

```
$ astrality
```

The result should be:

```
# Source: $XDG_CONFIG_HOME/config.ini

current_user=yourusername
```

This is probably a bit overwhelming. I recommend to just continue to the next page to get a more gentle introduction to these concepts.

1.4 Modules

1.4.1 What are modules?

Tasks to be performed by Astrality are grouped into so-called `modules`. These modules are used to define:

***Action blocks*:** A grouping of *actions* which is supposed to be performed at a specific time, such as "on Astrality startup", "on Astrality exit", or "on event".

Actions Tasks to be performed by Astrality, for example *compiling templates* or *running shell commands*.

Event listeners Event listeners can listen to predefined *events* and trigger the "on event" action block of the module.

You can easily enable and disable modules, making your configuration more modular.

1.4.2 How to define modules

There are two types of places where you can define your modules:

Directly in `$ASTRALITY_CONFIG_HOME/modules.yml`: Useful if you don't have too many modules, and you want to keep everything easily accessible in one file.

In a file named `modules.yml` within a *modules directory*: Useful if you have lots of modules, and want to separate them into separate directories with common responsibilities.

See the *documentation* for external modules for how to define modules this way.

You can use templating features in `modules.yml`, since they are compiled at startup with all context values defined in all `context.yml` files.

Hint: A useful configuration structure is to define modules with "global responsibilities" in `$ASTRALITY_CONFIG_HOME/modules.yml`, and group the remaining modules in seperate module directories by their categorical responsibilites (for example "terminals").

Here "global responsibility" means having the responsibility to satisfy the dependecies of several other modules, such as defining context values used in several templates, creating directories, or installing common dependencies.

Module definition syntax

Modules are formated as separate *dictionaries* placed at the root indentation level of "modules.yml". The key used will become the module name.

The simplest module, with no specific behaviour, is:

```
# Source: $ASTRALITY_CONFIG_HOME/modules.yml

my_module:
    enabled: true
```

Astrality skips parsing any modules which contain the option `enabled: false`. The default value of `enabled` is `true`, so you do not have to specify it.

Module dependencies

You can specify conditionals that must be satisfied in order to consider a module enabled. It can be useful if a module requires certain dependencies in order to work correctly

You can specify module requirements by setting the module option `requires` equal to a list of dictionaries containing one, or more, of the following keywords:

`env`: Environment variable specified as a string. The environment variable must be set in order to consider the module enabled.

`installed`: Program name specified as a string. The program name must be invokable through the command line, i.e. available through the `$PATH` environment variable. You can test this by typing `command -v program_name` in your shell.

`shell`: Shell command specified as a string. The shell command must return a 0 exit code (which defines success), in order to consider the module enabled.

If the shell command uses more than 1 second to return, it will be considered failed. You can change the default timeout by setting the *requires_timeout* configuration option.

You can also override the default timeout on a case-by-case basis by setting the `timeout` key to a numeric value (in seconds).

`module`: Module dependent on other module(s), specified with the same name syntax as with *enabled_modules*.

If a module is missing one or more module dependencies, it will be disabled, and an error will be logged.

All specified dependencies must be satisfied in order to enable the module.

For example, if your module depends on the `docker` shell command, another module named `docker-machine`, the environment variable `$ENABLE_DOCKER` being set, and "my_docker_container" existing, you can check this by setting the following requirements:

```
# Souce: $ASTRALITY_CONFIG_HOME/modules.yml

docker:
    requires:
        - installed: docker
```

(continues on next page)

(continued from previous page)

```
        - module: docker-machine
        - env: ENABLE_DOCKER
        - shell: '[ $(docker ps -a | grep my_docker_container) ]'
          timeout: 10 # seconds
```

Hint: `requires` can be useful if you want to use Astrality to manage your dotfiles. You can use module dependencies in order to only compile configuration templates to their respective directories if the dependent application is available on the system. This way, Astrality becomes a "conditional symlinker" for your dotfiles.

1.4.3 Action blocks

When you want to assign *tasks* for Astrality to perform, you have to define *when* to perform them. This is done by defining those `actions` in one of five available `action blocks`.

`on_setup`: Tasks to be performed only once and never again. Can be used for setting up dependencies.

Executed actions are written to `$XDG_DATA_HOME/astrality/setup.yml`, by default `$HOME/.local/share`. Execute `astrality --reset-setup module_name` if you want to re-execute a module's setup actions during the next run.

`on_startup`: Tasks to be performed when Astrality first starts up. Useful for compiling templates that don't need to change after they have been compiled.

Actions defined outside action blocks are considered to be part of this block.

`on_exit`: Tasks to be performed when you kill the Astrality process. Useful for cleaning up any unwanted clutter.

`on_event`: Tasks to be performed when the specified module `event listener` detects a new `event`. Useful for dynamic behaviour, periodic tasks, and templates that should change during runtime. The `on_event` block will never be triggered when no module event listener is defined. More on event listeners follows in *the next section*.

`on_modified`: Tasks to be performed when specific files are modified on disk. You specify a set of tasks to performed on a *per-file-basis*. Useful for quick feedback when editing template files.

> **Caution:** Only files within `$ASTRALITY_CONFIG_HOME/**/*` are observed for modifications.
>
> If this is an issue for you, please open a GitHub issue!

Demonstration of module action blocks:

```
module_name:
    ...startup actions (option 1)...

    on_setup:
        ...setup actions...

    on_startup:
        ...startup actions (option 2)...

    on_event:
```

(continues on next page)

(continued from previous page)

```
        ...event actions...

    on_exit:
        ...shutdow actions...

    on_modified:
        some/file/path:
            ...some/file/path modified actions...
```

Note: On Astrality startup, the `on_startup` event will be triggered, but **not** `on_event`. The `on_event` event will only be triggered when the `event listener` detects a new `event` *after* Astrality startup.

1.4.4 Actions

Actions are tasks for Astrality to perform, and are placed within *action blocks* in order to specify *when* to perform them. These are the available `action` types:

***import_context*:** Import a `context` section from a YAML formatted file. `context` variables are used as replacement values for placeholders in your *templates*. See *context* for more information.

***compile*:** Compile a specific template or template directory to a target path.

***copy*:** Copy a specific file or directory to a target path.

***symlink*:** Create symbolic link(s) pointing to a specific file or directory.

***stow*:** Combination of `compile` + `copy` or `compile` + `symlink`, bisected based on filename pattern of files within a content directory.

***run*:** Execute a shell command, possibly referring to any compiled template and/or the last detected *event* defined by the *module event listener*.

***trigger*:** Perform *all* actions specified within another *action block*. With other words, this action *appends* all the actions within another action block to the actions already specified in the action block. Useful for not having to repeat yourself when you want the same actions to be performed during different events.

Context imports

The simplest way to define *context values* is to just define their values in `$ASTRALITY_CONFIG_HOME/context.yml`. Those context values are available for insertion into all your templates.

But you can also import context values from arbitrary YAML files. Among other use cases, this allows you to:

- Split context definitions into separate files in order to clean up your configuration.
- Combine context imports with *on_event* blocks in order to dynamically change how templates compile. This allows quite complex behaviour.

Context imports are defined as a dictionary, or a list of dictionaries, if you need several imports. Use the `import_context` keyword in an *action block* of a module.

This is best explained with an example. Let us create a color schemes file:

```
# Source file: $ASTRALITY_CONFIG_HOME/modules/color_schemes/color_schemes.yml

gruvbox_dark:
    background: 282828
    foreground: ebdbb2

gruvbox_light:
    background: fbf1c7
    foreground: 3c3836
```

Then let us import the gruvbox *dark* color scheme into the "colors" *context* section:

```
# Source file: $ASTRALITY_CONFIG_HOME/modules.yml

color_scheme:
    on_startup:
        import_context:
            from_path: modules/color_schemes/color_schemes.yml
            from_section: gruvbox_dark
            to_section: colors
```

This is functionally equivalent to writing the following global context file:

```
# Source file: $ASTRALITY_CONFIG_HOME/context.yml

colors:
    background: 282828
    foreground: ebdbb2
```

Hint: You may wonder why you would want to use this kind of redirection when defininining context variables. The advantages are:

- You can now use `{{ colors.foreground }}` in all your templates instead of `{{ gruvbox_dark.foreground }}`. Since your templates do not know exactly *which* color scheme you are using, you can easily change it in the future by editing only one line in `modules.yml`.
- You can use `import_context` in a `on_event` action block in order to change your colorscheme based on the time of day. Perhaps you want to use "gruvbox light" during daylight, but change to "gruvbox dark" after dusk?

The available attributes for `import_context` are:

`from_path`: A YAML formatted file containing *context sections*.

`from_section`: ***[Optional]*** Which context section to import from the file specified in `from_path`.

If none is specified, all sections defined in `from_path` will be imported.

`to_section`: ***[Optional]*** What you want to name the imported context section. If this attribute is omitted, Astrality will use the same name as `from_section`.

This option will only have an effect if `from_section` is specified.

Compile templates

Template compilations are defined as a dictionary, or a list of dictionaries, under the `compile` keyword in an *action block* of a module.

Each template compilation action has the following available attributes:

`content`: Path to either a template file or template directory.

If `content` is a directory, Astrality will compile all templates recursively to the `target` directory, preserving the directory hierarchy.

`target`: ***[Optional]*** *Default:* Temporary file created by Astrality.

Path which specifies where to put the *compiled* template.

You can skip this option if you do not care where the compiled template is placed, and what it is named. You can still use the compiled result by writing `{template_path}` in the rest of your module. This placeholder will be replaced with the absolute path of the compiled template. You can for instance refer to the file in *a shell command*.

`include` ***[Optional]*** *Default:* `'(.+)'`

Regular expression defining which filenames that are considered to be templates. Useful when `content` is a directory which contains non-template files. By default Astrality will try to compile all files.

If you specify a capture group, astrality will use the captured string as the target filename. For example, `templates: 'template\.(.+)'` will match the file "template.kitty.conf" and rename the target to "kitty.conf".

Hint: You can test your regex here. Astrality uses the capture group with the greatest index.

`permissions`: ***[Optional]*** *Default:* Same permissions as the template file.

The file mode (i.e. permission bits) assigned to the *compiled* template. Given either as a string of octal permissions, such as `'755'`, or as a string of symbolic permissions, such as `'u+x'`. This option is passed to the linux shell command `chmod`. Refer to `chmod`'s manual for the full details on possible arguments.

Note: The permissions specified in the `permissions` option are applied *on top* of the default permissions copied from the template file.

For example, if the template's permissions are `rw-r--r-- (644)` and the value of `'ug+x'` is supplied for the `permissions` option, the `644` permissions will first be copied to the resulting compiled file and then `chmod ug+x` will be applied on top of that to give a resulting permission on the file of `rwxr-xr-- (754)`.

If an invalid value is supplied for the `permissions` option, only the default permissions are copied to the compiled file.

Here is an example:

```
# Source file: $ASTRALITY_CONFIG_HOME/modules.yml

desktop:
    compile:
        - content: modules/scripts/executable.sh.template
          target: ${XDG_CONFIG_HOME}/bin/executable.sh
          permissions: 0o555
        - content: modules/desktop/conky_module.template

    run:
```

(continues on next page)

(continued from previous page)

```
    - shell: conky -c {modules/desktop/conky_module.template}
    - shell: polybar bar
```

Notice that the shell command `conky -c {modules/desktop/conky_module.template}` is replaced with something like `conky -c /tmp/astrality/compiled.conky_module.template`.

Note: All relative file paths in modules are interpreted relative to the directory which contains "module.yml" which defines the module.

Symlink files

You can `symlink` a file or directory to a target destination. Directories will be recursively symlinked, leaving any non-conflicting files intact. The `symlink` action have the following available parameters.

`content:` The target of the symlinking, with other words a path to a file or directory with the actual file content.

If `content` is a directory, Astrality will create an identical directory hierarchy at the `target` directory path and create separate symlinks for each file in `content`.

`target:` Where to place the symlink(s).

> **Caution:** This is the *location* of the symlink, **not** where the symlink *points to*.

`include` ***[Optional]*** *Default:* `'(.+)'`

Regular expression restricting which filenames that should be symlinked. By default Astrality will try to symlink all files.

If you specify a capture group, astrality will use the captured string as the symlink name. For example, `include: 'symlink\.(.+)'` will match the file "symlink.wallpaper.jpeg" and rename the symlink to "wallpaper.jpeg".

Note: If you astrality encounters an existing **file** where it is supposed to place a symbolic link, it will rename the existing file to "filename.bak".

Copy files

You can `copy` a file or directory to a target destination. Directories will be recursively copied, leaving non-conflicting files at the target destination intact. The `copy` action have the following available parameters.

`content:` Where to copy *from*, with other words a path to a file or directory with existing content to be copied.

If `content` is a directory, Astrality will create an identical directory hierarchy at the `target` directory path and recursively copy all files.

`target:` A path specifying where to copy *to*. Any non-conflicting files at the target destination will be left alone.

include *[Optional]* *Default:* `'(.+)'`

Regular expression restricting which filenames that should be copied. By default Astrality will try to copy all files.

If you specify a capture group, astrality will use the captured string as the name for the copied file. For example, `include: 'copy\.(.+)'` will copy the file "copy.binary.blob" and rename the copy to "binary.blob".

permissions: *[Optional]* *Default:* Same permissions as the original file(s).

See *compilation permissions* for more information.

Stow a directory

Often you want to:

1. Move all content from a directory in your dotfile repository to a specific target directory, while...
2. Compiling any template according to a consistent naming scheme, and...
3. Symlink or copy the remaining files which are *not* templates.

The `stow` action type allows you to do just that! Stow has the following available parameters:

content: Path to a directory of mixed content, i.e. both templates and non-templates.

target: Path to directory where processed content should be placed. Templates will be compiled to `target`, and the remaining files will be treated according to the `non_templates` parameter.

templates: *[Optional]* *Default:* `'template\.(.+)'`

Regular expression restricting which filenames that should be compiled as templates. By default, Astrality will only compile files named "template.*" and rename the compilation target to "*".

See the compile action *include parameter* for more information.

non_templates: *[Optional]* *Default:* `'symlink'`

Accepts: `symlink`, `copy`, `ignore`

What to do with files that do not match the `templates` regex.

permissions: *[Optional]* *Default:* Same permissions as the original file(s).

See *compilation permissions* for more information.

Here is an example module which compiles all files matching the glob `$XDG_CONFIG_HOME/**/*.t`, and places the *compiled* template besides the template, but *without* the file extension ".t". It leaves all other files alone:

```
# Source file: $ASTRALITY_CONFIG_HOME/modules.yml

dotfiles:
    stow:
        content: $XDG_CONFIG_HOME
        target: $XDG_CONFIG_HOME
        templates: '(.+)\.t'
        non_templates: ignore
```

Run shell commands

You can instruct Astrality to run an arbitrary number of shell commands when different *action blocks* are triggered. Each shell command is specified as a dictionary. The shell command is specified as a string keyed to `shell`. Place the commands within a list under the `run` option of an *action block*. See the example below.

You can place the following placeholders within your shell commands

`{event}`: The last event detected by the *module event listener*.

`{template_path}`: Replaced with the absolute path of the *compiled* version of the template placed at the path `template_path`.

Example:

```
weekday_module:
    event_listener:
        type: weekday

    on_startup:
        run:
            - shell: 'notify-send "You just started Astrality, and the day is {event}"
↪'

    on_event:
        run:
            - shell: 'notify-send "It is now midnight, have a great {event}! I'm␣
↪creating a notes document for this day."'
            - shell: 'touch ~/notes/notes_for_{event}.txt'

    on_exit:
        run:
            - shell: 'echo "Deleting today's notes!"'
            - shell: 'rm ~/notes/notes_for_{event}.txt'
```

You can actually place these placeholders in any action type's string values. Placeholders are replaced at runtime every time an action is triggered.

Warning: `template/path` must be compiled when an action type with a `{template/path}` placeholder is executed. Otherwise, Astrality does not know what to replace the placeholder with, so it will leave it alone and log an error instead.

Trigger action blocks

From one *action block* you can trigger another action block by specifying a `trigger` action.

Each trigger option is a dictionary with a mandatory `block` key, on of `on_startup`, `on_event`, `on_exit`, or `on_modified`. In the case of setting `block: on_modified`, you have to specify an additional `path` key indicating which file modification block you want to trigger.

An example of a module using `trigger` actions:

```
module_using_triggers:
     event_listener:
         type: weekday
```

(continues on next page)

(continued from previous page)

```
    on_startup:
        run:
            - shell: startup_command

        trigger:
            - block: on_event

    on_event:
        import_context:
            - from_path: contexts/A.yml
              from_section: '{event}'
              to_section: a_stuff

        trigger:
            - block: on_modified
              path: templates/templateA

    on_modified:
        templates/A.template:
            compile:
                content: templates/A.template

            run: shell_command_dependent_on_templateA
```

This is equivalent to writing the following module:

```
module_using_triggers:
    event_listener:
        type: weekday

    on_startup:
        import_context:
            - from_path: contexts/A.yml
              from_section: '{event}'
              to_section: a_stuff

        compile:
            content: templates/templateA

        run:
            - shell: startup_command
            - shell: shell_command_dependent_on_templateA

    on_event:
        import_context:
            from_path: contexts/A.yml
            from_section: '{event}'
            to_section: a_stuff

        compile:
            content: templateA

        run:
            - shell: shell_command_dependent_on_templateA

    on_modified:
        templates/templateA:
```

(continues on next page)

(continued from previous page)

```
            compile:
                content: templates/templateA

            run:
                - shell: shell_command_dependent_on_templateA
```

Hint: You can use `trigger: on_event` in the `on_startup` block in order to consider the event detected on Astrality startup as a new `event`.

The `trigger` action can also help you reduce the degree of repetition in your configuration.

The execution order of module actions

The order of action execution is as follows:

1. *context_import* for each module.
2. *symlink* for each module.
3. *copy* for each module.
4. *compile* for each module.
5. *stow* for each module.
6. *run* for each module.

Modules are iterated over from top to bottom such that they appear in `modules.yml`. This ensures the following invariants:

- When you compile templates, all `context` imports have been performed, and are available for placeholder substitution.
- When you run shell commands, all (non-)templates have been compiled/copied/symlinked, and are available for reference.

1.4.5 Global configuration options for modules

Global configuration options for all your modules are specified in `$ASTRALITY_CONFIG_HOME/astrality.yml` within a dictionary named `modules` at root indentation, i.e.:

```
# Source file: $ASTRALITY_CONFIG_HOME/astrality.yml

modules:
    option1: value1
    option2: value2
    ...
```

Available modules configuration options:

`requires_timeout`: *Default:* `1`

Determines how long Astrality waits for *module requirements* to exit successfully, given in seconds. If the requirement times out, it will be considered failed.

Useful when requirements are costly to determine, but you still do not want them to time out.

`run_timeout`: *Default:* `0`

Determines how long Astrality waits for module *run actions* to exit, given in seconds.

Useful when you are dependent on shell commands running sequantially.

`reprocess_modified_files`: *Default:* `false`

If enabled, Astrality will watch for file modifications in `$ASTRALITY_CONFIG_HOME`. All files that have been compiled or copied to a destination will be recompiled or recopied if they are modified.

Hint: You can have more fine-grained control over exactly *what* happens when a file is modified by using the `on_modified` *module event*. This way you can run shell commands, import context values, and compile arbitrary templates when specific files are modified on disk.

Caution: At the moment, Astrality only watches for file changes recursively within `$ASTRALITY_CONFIG_HOME`.

`modules_directory`: *default:* `modules`

Where Astrality looks for externally defined configurations directories.

`enabled_modules`: *default:*

```
enabled_modules:
    - name: '*'
    - name: '*::*'
```

A list of modules which you want Astrality to use. By default, Astrality enables all defined modules.

Specifying `enabled_modules` allows you to define a module without necessarily using it, making configuration switching easy.

Module defined in "*$ASTRALITY_CONFIG_HOME/modules.yml*": `name: name_of_module`

Module defined in "*<modules_directory>/dir_name/modules.yml*": `name: dir_name::name_of_module`

Module defined at "*github.com/<user>/<repo>/blob/master/modules.yml*": `name: github::<user>/<repo>::name_of_module`

You can also use wildcards when specifying enabled modules:

- `name: '*'` enables all modules defined in: `$ASTRALITY_CONFIG_HOME/modules.yml`.
- `name: 'text_editors::*` enables all modules defined in: `$ASTRALITY_CONFIG_HOME/<modules_directory>/text_editors/modules.yml`.
- `name: '*::*` enables all modules defined in: `$ASTRALITY_CONFIG_HOME/<modules_directory>/*/modules.yml`.

1.4.6 Module subdirectories

You can define "external modules" in files named `modules.yml` placed within separate subdirectories of your *modules directory*. You can also place `context.yml` within these directories, and the context values will become available for compilation in all templates.

Astrality compiles enabled `modules.yml` files with context from all enabled `context.yml` files before parsing it. This allows you to modify the behaviour of modules based on context, useful if you want to offer configuration options for modules.

1. Define your modules in `$ASTRALITY_CONFIG_HOME/<modules_directory>/directory/modules.yml`.
2. *Enable* modules from this config file by appending `name: directory::module_name` to `enabled_modules`. Alternatively, you can enable *all* modules defined in a module directory by appending `name: directory::*` instead.

By default, all module subdirectories are enabled.

Context values defined in `context.yml` have preference above context values defined in module subdirectories, allowing you to define default context values, while still allowing others to override these values.

Caution: All relative paths and shell commands in external modules are interpreted relative to the external module directory, not `$ASTRALITY_CONFIG_HOME`. This way it is more portable between different configurations.

1.4.7 GitHub modules

You can share a module directory with others by publishing the module subdirectory to GitHub. Just define `modules.yml` at the repository root, i.e. where `.git` exists, and include any dependent files within the repository.

Others can fetch your module by appending `name: github::<your_github_username>/<repository>` to `enabled_modules`.

For example enabling the module named `module_name` defined in `modules.yml` in the repository at https://github.com/username/repository:

```
modules:
    enabled_modules:
        - name: github::username/repository::module_name
```

Astrality will automatically clone the module on first-time startup, placing it within `$XDG_DATA_HOME/astrality/repositories/github/username/repository`. If you want to automatically update the GitHub module, you can specify `autoupdate: true`:

```
modules:
    enabled_modules:
        - name: github::username/repository::module_name
          autoupdate: true
```

If `module_name` is not specified, all modules will be enabled:

```
modules:
    enabled_modules:
        - name: github::username/repository
          autoupdate: true
```

1.5 Event listeners

1.5.1 What are event listeners?

`Event listeners` provide you with the ability to keep track of certain events, and change module behaviour accordingly. A short summation of event listeners:

1. Event listeners are specified on a *per-module-basis*.
2. There are different `types` of event listeners.
3. Event listeners determine exactly *when* the *actions* you specify within a module's *on_event action block* are executed.
4. Event listeners are optional, you can write valid modules without specifying one. Actions specified within the `on_event` action block will never be executed when no module event listener is specified.
5. Event listeners provide the module with the `{event}` placeholder when specifying `actions`. It is replaced by the current `event` at runtime. The replacement value is specific for that specific event listener's type, and dynamically changes according to rules set by the event listener.

1.5.2 What are event listeners used for?

Event listeners provide you with the tools needed for *dynamic* module behaviour. Sometimes you want a module to *execute* different shell commands, and/or *compile* templates with *different context values*, depending on exactly *when* those *actions* are performed.

1.5.3 How to set a module event listener

Module event listeners are defined within the *module block* it is supposed to provide functionality for. The syntax is as follows:

```
some_dynamic_module:
    event_listener:
        type: type_of_event_listener

        option1: whatever
        option2: something
        ...
```

Most event listeners provide you with additional options in order to tweak their behaviour. These are specified at the same indentation level as the event listener type.

1.5.4 Events

Module event listeners keep track of some type of `event` and trigger the `on_event` *action block* whenever it detects a *new* event. You can refer to the current event in your module actions with the `{event}` placeholder.

Caution: When you use placeholders, you must take care that the placeholder is not interpreted as a YAML *dictionary* instead of a *string*. The following will not work as intended:

```
some_option: {event}
```

This is interpreted as the dictionary `{'event': None}`. In this case you must mark the option explicitly as a string:

```
some_option: '{event}'
```

Using quotes is not necessary when the placeholder is part of a greater string. This works:

```
some_option: echo {event}
```

An example using events

The use of `events` in modules is best explained with an example. Please take a look at *this example* using the `weekday` event listener in order to set a separate desktop wallpaper for each day of the week.

1.5.5 Event listener types

Here is a list of all available Astrality module event listeners and their configuration options. If what you need is not available, feel free to open an issue with a event listener request!

Daylight

Description Keeps track of the daylight at a specific location, i.e. if the sun is above the horizon or not.

Specifier `type: daylight`

Events `day`, `night`

Table 1: Configuration options

Option	Default	Description
latitude	0	Latitude coordinate point of your location.
longitude	0	Longitude coordinate point of your location.
elevation	0	Height above sea level at your location.

These coordinates can be obtained from this website.

Example configuration

```
daylight_module:
    event_listener:
        type: daylight

        latitude: 63.446827
        longitude: 10.421906
```

Solar

Description Keeps track of the sun's position in the sky at a given location.

Specifier `type: solar`

Events `sunrise`, `morning`, `afternoon`, `sunset`, `night`

Table 2: Configuration options

Option	Default	Description
latitude	0	Latitude coordinate point of your location.
longitude	0	Longitude coordinate point of your location.
elevation	0	Height above sea level at your location.

These coordinates can be obtained from this website.

Example configuration

```
solar_module:
    event_listener:
        type: solar

        latitude: 63.446827
        longitude: 10.421906
```

Static

Description An event listener which never changes its event. This is the default event listener for modules.

Specifier `type: static`

Events `static`

No configuration options are available for the static event listener.

Example configuration

```
static_module:
    ...
```

Time of day

Description Keeps track of a specific time interval for each day of the week. Useful for tracking when you are at work.

Specifier `type: time_of_day`

Events `on`, `off`

Table 3: Configuration options

Option	Default	Description
monday	`'09:00-17:00'`	The time of day that is considered 'on'.
tuesday	`'09:00-17:00'`	The time of day that is considered 'on'.
wednesday	`'09:00-17:00'`	The time of day that is considered 'on'.
thursday	`'09:00-17:00'`	The time of day that is considered 'on'.
friday	`'09:00-17:00'`	The time of day that is considered 'on'.
saturday	`''`	The time of day that is considered 'on'.
sunday	`''`	The time of day that is considered 'on'.

Example configuration

```
european_tue_to_sat_work_week:
    event_listener:
        type: time_of_day
        monday: ''
        tuesday: '08:00-16:00'
        wednesday: '08:00-16:00'
        thursday: '08:00-16:00'
        friday: '08:00-16:00'
        saturday: '08:00-16:00'
```

Weekday

Description Keeps track of the weekdays.

Specifier `type: weekday`

Events `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday`, `sunday`

No configuration options are available for the weekday event listener.

Example configuration

```
weekday_module:
    event_listener:
        type: weekday
```

Periodic

Description Keeps track of constant length time intervals.

Specifier `type: periodic`

Events `0`, `1`, `2`, `3`, and so on...

Table 4: Configuration options

Option	Default	Description
seconds	0	Number of seconds between each period.
minutes	0	Number of minutes between each period.
hours	0	Number of hours between each period.
days	0	Number of days between each period.

If the configured time interval is of zero length, Astrality uses `hours: 1` instead.

Example configuration

```
periodic_module:
    event_listener:
        type: periodic
        hours: 8
```

1.6 Tutorial

1.6.1 Writing an application configuration template

Motivation

In order to use an application on a UNIX system, there are often several tasks that need to be performed before doing so.

- Determine if the application should be installed on the system in the first place.
- Install the package.
- Place the default configuration file for the application in its appropriate location.
- Tweak configuration paramateres according to the host environment and personal preferences.
- Start any required background processes on startup.

Preferably you would want to do this work once, and easily deploy such configurations across different systems. Astrality enables you to manage such tasks in a reproducible and sharable way, grouping those tasks together into a single configuration.

For the sake of example, we will use Astrality to manage the configuration of polybar, a status line application for Linux. Hopefully, we will be able to demonstrate that the end result is a configuration that is easy to tweak, re-deploy, and share with others.

The task

In most cases, you will have an existing configuration to start from. In this case it will be the default configuration shipped with polybar. Here is a small extract of this default configuration file:

Listing 1: ~/.config/polybar/config

```
[bar/example]
;monitor = ${env:MONITOR:HDMI-1}

font-0 = fixed:pixelsize=10;1
font-1 = unifont:fontformat=truetype:size=8:antialias=false;0
font-2 = siji:pixelsize=10;1

[module/wlan]
type = internal/network
interface = wlp3s0
interval = 3.0
```

This extract contains the two types of configuration types one usually wants to change:

- **Personal preference** - The three main fonts used in the status bar.
- **Host environment** - The wlan interface identifier.

Create a polybar module

We will create a Astrality module which is responsible for the management of all things related polybar.

Modules can by defined in either `~/.config/astrality/modules.yml` or `~/.config/astrality/modules/<module_group>/modules.yml`. You can tweak these locations to your preferences by setting

$ASTRALITY_CONFIG_HOME and/or by setting the *modules_directory* config option. For now, we will create a `statusbars` module group in the latter default location.

Let's start by creating a seperate directory for this module group:

```
$ mkdir -p ~/.config/astrality/modules/statusbars && cd $_
```

We will also move the default configuration file to this folder to keep everything in one place. This file will be used as a template for compilation so we will prefix the filename with `template.` to make this clear:

```
$ mv ~/.config/polybar/config template.config
```

We will *define a module* named `polybar` which *compiles* this template to the previous location:

Listing 2: ~/.config/astrality/modules/statusbars/modules.yml

```
polybar:
  compile:
    content: "template.config"
    target: "~/.config/polybar/config"
```

You can also instruct Astrality to *copy* or *symlink*, optionally recursively. See the *actions documentation* for more information.

We can now compile this template by running: `astrality -m statusbars::polybar`, or alternatively just `astrality`, as all defined modules are enabled by default. An optional `--dry-run` flag is supported if you want to safely check which actions will be executed.

At this point this is nothing more than a glorified copy script, but we can now start to insert Jinja2 templating syntax into this file.

Writing the template with context placeholders

We can start by defining some context values which we want to insert into our template:

Listing 3: ~/.config/astrality/modules/statusbars/context.yml

```
statusbar:
  font:
    size: 16

    1: "FuraCode Nerd Font"
    2: "FuraCode Mono Nerd Font"

host:
  interfaces:
    wlan:
      handle: "wlp3s0"

    ethernet:
      handle: "eno0"
```

And now let's use *placeholders* in the template where these values should be inserted:

Listing 4: ~/.config/astrality/modules/statusbars/template.config

```
[bar/example]
;monitor = ${env:MONITOR:HDMI-1}

font-0 = {{ statusbar.font.1 }}:pixelsize={{ statusbar.font.size }};1
font-1 = {{ statusbar.font.2 }}:fontformat=truetype:size={{ statusbar.font.size }}
↪:antialias=false;0
font-2 = {{ statusbar.font.3 }}:pixelsize={{ statusbar.font.size }};1

[module/wlan]
type = internal/network
interface = {{ host.interfaces.wlan.handle }}
interval = 3.0
```

The compilation target will replace these placeholders with the placeholders defined in `context.yml`, and you can check the result by running `astrality -m statusbars::polybar` again.

This extracts the configuration options which are of interest into a much more succinct file, enabling us to tweak it easily. The same placeholders can be used in other templates, which can make switching between different status bars more consistent, for instance. There are also other benefits related to *sharing* modules, which we will come back to later.

Hint: You may have noticed that we only defined *two* fonts in `context.yml`, while using *three* fonts in the template, thinking that the use of `{{ statusbar.font.3 }}` is undefined. But for numeric context keys, astrality will fall back to the greatest number available.

With other words: `statusbar.font.3 -> statusbar.font.2`.

This allows us to specify an additional font in the future if we want to.

More information can be found in the *templating documentation*.

Expanding the module

Starting polybar

Currently, Astrality only compiles the template and quits. We can do better! First, we can instruct Astrality to start polybar after having compiled the template:

Listing 5: ~/.config/astrality/modules/statusbars/modules.yml

```
polybar:
  compile:
    content: "template.config"
    target: "~/.config/polybar/config"

  run:
    shell: "polybar --config=~/.config/polybar/config example"
```

We can now compile the template *and* start polybar by running `astrality -m statusbars::polybar`.

When using the `--config` polybar flag, we do not actually care exactly *where* the compiled template is saved, as long as we can provide the compilation path to polybar. We can therefore skip specifying the `target` for the compilation and instead use `{template.config}` in the shell command. This placeholder will be replaced with the file path to the compiled template.

Listing 6: ~/.config/astrality/modules/statusbars/modules.yml

```
polybar:
  compile:
    content: "template.config"

  run:
    shell: "polybar --config={template.config} example"
```

This will reduce additional clutter on our filesystem and prevent overwriting any existing files. This unique compilation target will make the module easier to share with other, a topic which we will come back to soon.

We can also kill potentially existing polybar processes before starting the new one:

Listing 7: ~/.config/astrality/modules/statusbars/modules.yml

```
polybar:
  compile:
    content: "template.config"

  run:
    - shell: "killall -q polybar"
    - shell: "polybar --config={template.config} example"
```

See *the run action* for more information regarding the execution of shell commands within Astrality.

Requirements

By default, all modules defined in subdirectories of `~/.config/astrality/modules` will be enabled. See *enabled modules documentation* for how to gain more fine-grained control.

We can add *additional* constrains for when we consider a module enabled. In this case, we can require polybar to be installed on the system. If polybar is *not* installed, Astrality will skip any further module action and log a warning.

Listing 8: ~/.config/astrality/modules/statusbars/modules.yml

```
polybar:
  requires:
    installed: polybar

  compile:
    content: "template.config"

  run:
    - shell: "killall -q polybar"
    - shell: "polybar --config={template.config} example"
```

You can also add requirements related to environmet variables, shell command exit codes, and other astrality modules. Alternatively, you can define actions within an `on_setup` block to install such dependencies once, and only once. See *module dependencies* and *action blocks* for more information.

Sharing your module

Publish to GitHub

You can easily share an Astrality module by publishing the module directory to GitHub as a repository. `modules.yml` and `context.yml` must be located at the root level of the repository. An example module repository can be found here.

Fetch module from GitHub

Let us assume that your GitHub username is `username` and you published the statusbars module directory as part of a repository named `statusbars`. Other people can now try your status bar configuration by running:

```
$ astrality -m github::username/statusbars::polybar
```

Astrality will automatically clone the repository and execute the module's actions. They will be able to quickly judge if your specific configuration is to their taste or not.

Overriding context values

The context values used in the polybar template was earlier defined in `context.yml` within the module directory. Any such context key can be overwritten by defining the same key in the global context store located at `~/.config/astrality/context.yml`.

This allows anybody to specify *their* favorite font and correct WLAN interface handle, while still using *your* polybar configuration.

It is this that makes Astrality modules much more sharable across different preferences and host environments. The `context.yml` clearly stipulates which parameters which someone (including you) probably want to change at some time.

If someone want specify their correct interfaces, while keeping your specified font, they can define the following global context items (all without taking a deep-dive into your configuration):

Listing 9: ~/.config/astrality/context.yml

```
host:
  interfaces:
    wlan:
      handle: "wlp3s1"

    ethernet:
      handle: "eno2"
```

Permanently add a GitHub module

If the user decides to keep the module in use, they can add `github::username/statusbars::polybar` to the `enabled_modules` section in `~/.config/astrality/astrality.yml`. It will be added to any existing modules when executing `astrality` in the shell.

Clean up files created by a module

You can easily clean up any files created by a module of any type, restoring any overwritten files in the process. Use the `--cleanup` flag with the same name you would use to enable the module. For example:

```
$ astrality --cleanup github::username/statusbars::polybar
```

If a module has overwritten a valuable file, you can use this option to restore it. It also makes it easy to remove configuration files for applications you no longer use! You can also try a new module with the `--dry-run` flag to safely check which actions that will be executed.

1.6.2 Managing dotfiles with templates

It is relatively common to organize all configuration files in a "dotfiles" repository. How you structure such a repository comes down to personal preference. We would like to use the templating capabilities of Astrality without making any changes to our existing dotfiles hierarchy. This is relatively easy!

Let us start by managing the files located in `$XDG_CONFIG_HOME`, where most configuration files reside. The default value of this environment variable is "~/.config". We will create an Astrality module which automatically detects files named "template.whatever", and compile it to "whatever". This way you can easily write new templates without having to add new configuration in order to compile them.

```
# ~/.config/astrality/modules.yml

dotfiles:
    compile:
        content: $XDG_CONFIG_HOME
        target: $XDG_CONFIG_HOME
        include: 'template\.(.+)'
```

Let us go through the module configuration step-by-step:

- We use the `compile` action type, as we are only interested in compiling templates at the moment.
- We set both the content and target to be `$XDG_CONFIG_HOME`, compiling any template to the same directory as the template.
- We only want to compile template filenames which matches the regular expression `template\.(.+)`.
- The regex capture group in `template\.(.+)` specifies that everything appearing after "template." should be used as the *compiled* target filename.

We can now compile all such templates within *$XDG_CONFIG_HOME* by running `astrality` from the shell. Before doing so, it is recommended to run `astrality --dry-run` to see which actions that will be performed.

But we would like to *automatically* recompile templates when we modify them or create new ones. You can achieve this by enabling `reprocess_modified_files` in `astrality.yml`:

```
# ~/.config/astrality/astrality.yml

config/modules:
    reprocess_modified_files: true
```

Astrality will automatically recompile any modified templates as long as it runs as a background process.

Let us continue by managing a more complicated dotfiles repository. Most people create a separate repository containing *all* their configuration files, not only `$XDG_CONFIG_HOME`. The repository is then cloned to something like `~/.dotfiles`, the contents of which is symlinked or copied to separate locations, `$HOME`, `$XDG_CONFIG_HOME`, `$/etc` on so on. You can do all of this with Astrality.

For demonstration purposes, let us assume that the templates within "~/.dotfiles/home" should be compiled to "~", and "~/.dotfiles/etc" to "/etc", while non-templates should be symlinked instead. This combination of *symlink* and *compile* actions can be done with the *stow* action.

Move `modules.yml` and `astrality.yml` to the root of your dotfiles repository. Set `export ASTRALITY_CONFIG_HOME=~/.dotfiles`. Finally, modify the dotfiles module accordingly:

```
# ~/.dotfiles/modules.yml

dotfiles:
    stow:
        - content: home
            target: ~
            templates: 'template\.(.+)'
            non_templates: symlink

        - content: etc
            target: /etc
            templates: 'template\.(.+)'
            non_templates: symlink
```

`templates: 'template\.(.+)'` and `non_templates: symlink` are actually the default options for the stow action, so we could have skipped specifying them altogether. Alternatively, you can specify `non_templates: copy`.

You can now start to write all your configuration files as templates instead, using placeholders for secret API keys or configuration values that change between machines, and much much more.

1.6.3 A module using events

Let us explore the use of `events` with an example: we want to use a different desktop wallpaper for each day of the week.

The `weekday` event listener type keeps track of the following events: `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday`, and `sunday`.

After having found seven fitting wallpapers, we name them according to the weekday we want to use them, and place them in `$ASTRALITY_CONFIG_HOME/modules/weekday_wallpaper/`:

```
$ ls -l $ASTRALITY_CONFIG_HOME/modules/weekday_wallpaper

monday.jpeg
tuesday.jpg
wednesday.png
thursday.tiff
friday.gif
saturday.jpeg
sunday.jpeg
```

Now we need to create a module with a `weekday` event listener in `modules.yml`:

```
weekday_wallpaper:
    event_listener:
        type: weekday
```

We also need a way of setting the desktop wallpaper from the shell. Here we are going to use the feh shell utility. Alternatively, on MacOS, we can use this script. After having installed `feh`, we can use it to set the appropriate wallpaper on Astrality startup:

```
weekday_wallpaper:
    event_listener:
```

(continues on next page)

(continued from previous page)

```
        type: weekday

    on_startup:
        run:
            - shell: feh --bg-fill modules/weekday_wallpaper/{event}.*
```

Now Astrality will set the appropriate wallpaper on startup. We still have a small bug in our module. If you do not restart Astrality the next day, yesterday's wallpaper will still be in use. We can fix this by changing the wallpaper every time the weekday *changes* by listening for the weekday event.

```
weekday_wallpaper:
    event_listener:
        type: weekday

    on_startup:
        run:
            - shell: feh --bg-fill modules/weekday_wallpaper/{event}.*

    on_event:
        run:
            - shell: feh --bg-fill modules/weekday_wallpaper/{event}.*
```

Or, alternatively, we can just *trigger* the `on_startup` action block when the event changes:

```
weekday_wallpaper:
    event_listener:
        type: weekday

    on_startup:
        run:
            - shell: feh --bg-fill modules/weekday_wallpaper/{event}.*

    on_event:
        trigger:
            - block: on_startup
```

1.7 Example configuration

Here is an example configuration of `$ASTRALITY_CONFIG_HOME`, which you can copy as a starting point by running `astrality --create-example-config`.

First the global configuration options:

Listing 10: $ASTRALITY_CONFIG_HOME/astrality.yml

```
astrality:
    # If hot_reload_config is enabled, modifications to this file automatically
    # runs:
    #       1) exit actions from the old configuration
    #       2) startup actions from the new configuration
    # Requires restart if enabled
    hot_reload_config: true

    # You can delay astrality on startup. The delay is given in seconds.
```

(continues on next page)

(continued from previous page)

```
    startup_delay: 0


modules:
    # Modules can require successfull shell commands (non-zero exit codes) in
    # order to be enabled. You can specify the timeout for such checks, given
    # in seconds.
    requires_timeout: 1

    # Astrality can wait for shell commands to complete in their specified
    # order. You can set the number of seconds Astrality waits for the shell
    # commands to exit.
    run_timeout: 0

    # Modified templates can be automatically recompiled. This also includes
    # files that have been copied to a target destination.
    reprocess_modified_files: true

    # There are two possible ways to define modules. Either in this file, as
    # shown further below, or in separate external module directories within the
    # following specified directory, relatively interpreted as:
    # $ASTRALITY_CONFIG_HOME/modules
    modules_directory: modules

    # You enable modules by specifying the <name> of each module.
    # Modules defined in <modules_directory>/<subdirectory>/config.yml are
    # enabled by writing name: <subdirectory>::<name>
    #
    # '*' enables all modules in this file, '*::*' enables all modules defined
    # in subdirectories of <modules_directory>.
    enabled_modules:
        # Module defined in this file
        - name: polybar::*
        - name: terminals

        # All modules defined in <modules_directory>/solar_desktop/config.yml
        - name: solar_desktop::*

        # Module defined at https://github.com/jakobgm/color-schemes.astrality
        - name: github::jakobgm/color-schemes.astrality
          autoupdate: true  # Fetch new color schemes as they are added
```

Then some example modules:

Listing 11: $ASTRALITY_CONFIG_HOME/modules.yml

```
dotfiles:
    # This module automatically compiles all filenames within $XDG_CONFIG_HOME
    # prefixed with 'template.'. It removes the prefix for the compile target,
    # placing it in the same directory as the template.
    compile:
        content: $XDG_CONFIG_HOME
        target: $XDG_CONFIG_HOME
        include: 'template\.(.+)'


terminals:
```

(continues on next page)

(continued from previous page)

```
    # By default, this module is not enabled, since it overwrites possibly
    # pre-existing configuration files. Enable it in config/modules.
    #
    # This module uses the color scheme context syntax from:
    # github::jakobgm/color-schemes.astrality
    # And the color scheme can be changed in context/color_schemes_config
    #
    # It makes it easy to change color schemes for all your terminals at the
    # same time.
    #
    # Terminals:
    # Alacritty: https://github.com/jwilm/alacritty
    # Kitty: https://github.com/kovidgoyal/kitty

    requires:
        - installed: 'alacritty'
        - installed: 'kitty'

    compile:
        - content: modules/terminals/alacritty.yml.template
          target: {{ env.XDG_CONFIG_HOME }}/alacritty/alacritty.yml
        - content: modules/terminals/kitty.conf.template
          target: {{ env.XDG_CONFIG_HOME }}/kitty/kitty.conf
```

Finally some useful context values to be used in templates:

Listing 12: $ASTRALITY_CONFIG_HOME/context.yml

```
host:
    # Here we define some context values which often change between host
    # computers, and are therefore practical to use in our templates.
    displays:
        # All displays defined here are used in the polybar module. It creates
        # one bar for each of the display handles, where the bar identifier is
        # the same as the display handle. This way you can start a polybar for
        # the primary screen by running:
        #     polybar --config {modules/polybar/config.template} HDMI2
        primary:
            handle: HDMI2
            dpi: 96

        secondary:
            handle: eDP1
            dpi: 96


    interfaces:
        wlan:
            # You can also use command substitution in order to insert the
            # standard output of a shell command into a configuration option.
            #
            # This is also used by the polybar template to point the
            # wireless-internet polybar module to the correct interface.
            handle: {{ 'iwconfig 2>/dev/null | grep -o "^\w*"' | shell }}

        ethernet:
            handle: eno0
```

(continues on next page)

(continued from previous page)

```
    commands:
        # Here we define some commands that might change between hosts
        # with different stacks, i.e. systemd vs init, or wayland vs Xorg
        shutdown: systemctl poweroff -i
        reboot: systemctl reboot -i

        # In order to insert our global IP into a template, we can now do:
        # host.commands.global_ip | shell within placeholder delimiters
        global_ip: 'wget http://checkip.dyndns.org/ -O - -o /dev/null | cut -d: -f 2␣
↪| cut -d\< -f 1 | xargs'


fonts:
    # Here we define some context values for fonts that we want to use in
    # several different configurations, another common use case for context
    # values in templates.

    # You can use integer indexed variables in order to have fallback values.
    # If fonts:4/5/6 on so on is used in a template, but it is not
    # defined, it will be replaced with ast:fonts:3 instead. This is
    # useful when you dont want to assume how many fonts you want to use when
    # you write your templates.
    #
    # Here we define the main fonts used across our applications. Where 1
    # is the primary font, 2 the secondary font, and so on.

    # These fonts can be installed here: https://nerdfonts.com/
    1:
        name: FuraCode Nerd Font
        size: 8

    2:
        name: FuraCode Nerd Font Mono
        size: 8

    3:
        name: RobotoMono Nerd Font
        size: 8

    # We also add some configurations which are specific for some application
    # types.
    terminal:
        size: 10

    status_bar:
        size: 8


color_schemes_config:
    # These are context values used by the GitHub module:
    # github.com/jakobgm/color-schemes.astrality
    # See the README of this repository for more information.

    # Enable the following color scheme:
    enabled: gruvbox_dark
```

(continues on next page)

(continued from previous page)

```
    # Import the color scheme into the following context section
    context_section: colors
```

1.8 Tips and Tricks

1.8.1 Configuration of other applications

i3wm

You probably want to automatically start Astrality on startup. Here is an example for those who use the i3 tiling window manager.

Add the following line to `$XDG_CONFIG_HOME/i3/config`:

```
exec --no-startup-id "astrality"
```

Compton

If you are using the compton compositor, and want to use the conky `modules` included in the example configuration, you should disable any shadows and dims which could be applied to the conky desktop modules. Here is an example compton configuration which you should place at `$XDG_CONFIG_HOME/compton/compton.conf`:

```
inactive-dim = 0.1;
shadow = true;
shadow-exclude = [
    "! name~=''",
    "class_g = 'Conky'"
    ]
mark-ovredir-focused = true;
```

1.9 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

1.9.1 [1.1.1] - 2018-11-27

Fixed

- Astrality now cleans up directories created by the *compile*, *stow*, *symlink* and *copy* actions when modules are cleaned up with the *–cleanup* flag.

1.9.2 [1.1.0] - 2018-06-24

Changed

- Astrality no longer reverts to using the example configuration when `ASTRALITY_CONFIG_HOME/astrality.yml` does not exist. Default values are are used instead, and a warning is logged. Use `--create-example-config` to get create the example configuration instead.
- GitHub modules are now cloned to `$XDG_DATA_HOME/astrality/repositories/github` instead of `$ASTRALITY_CONFIG_HOME/<modules_directory>`.

1.9.3 [1.0.3] - 2018-06-17

Changed

- Astrality is now marked as "production/stable" on PyPI.

Fixed

- Fixed bug which caused `$ASTRALITY_LOGGING_LEVEL` and `astrality -l <logging_level>` to be ignored.
- Astrality now catches errors caused by starting the file system watcher. It logs the error and continues on without watching the error in such a case.

1.9.4 [1.0.2] - 2018-05-25

Fixed

- Fixed lint errors in documentation which caused incorrect rendering on PyPI.

1.9.5 [1.0.1] - 2018-05-24

Fixed

- Added missing dependency `python-dateutil` to `setup.py`.

1.9.6 [1.0.0] - 2018-05-24

Added

- New `symlink` action type.
- New `copy` action type.
- New `stow` action type. This action allows you to either compile+symlink or compile+copy, bisecting a directory based on filename regular expression matching.
- You can now compile all templates recursively within a directory. Just set `content` to a directory path. `target` must be a directory as well, and the relative file hierarchy is preserved.
- You can now specify which filenames are considered templates when compiling directories recursively.

- Template target filenames can now be renamed by specifying a regular expression capture group.
- Non-template files can now be either symlinked, copied, or ignored.
- The run action now supports `timeout` option, in order to set `run_timeout` on command-by-command basis.
- `compile` actions now support an optional `permissions` field for setting the permissions of the compiled template. It allows setting octal values such as `'755'`, and uses the UNIX `chmod` API.
- Module requirements can now specify required programs and environment variables by using the dictionary keys `installed` and `env` respectively.
- You can now set `requires` timeout on a case-by-case basis.
- Add new `--module` CLI flag for running specific modules.
- `on_startup` blocks can now optionally be implicitly defined at the root indentation level in the module.
- You can now run astrality with `--dry-run` in order to check which actions that will be executed.
- Modules can now depend on other modules with the `module` requires keyword.
- Modules can now place action in a `setup` block, only to be executed once.
- You can now execute `astrality --reset-setup module_name` in order to clear executed module setup actions.
- Files created by `compile`, `copy`, `stow`, and `symlink` actions are now persisted and cleaned up when executing `astrality --cleanup MODULE`. Files that are overwritten by Astrality are backed up and restored on clean up.

Changed

- `astrality.yml` has now been split into three separate files: `astrality.yml` for global configuration options, `modules.yml` for global modules, and `context.yml` for global context.
- Directory module config file `config.yml` has been renamed and split into `modules.yml` and `context.yml`. See point above.
- The `run` module action is now a dictionary instead of a string. This enables us to support additional future options, such as `timeout`. Now you specify the shell command to be run as a string value keyed to `shell`.

  Old syntax:

```
run:
    - command1
    - command2
```

  New syntax:

```
run:
    - shell: command1
    - shell: command2
```

- The `trigger` module action is now a dictionary instead of a string. Now you specify the block to be triggered as a string value keyed to `block`. `on_modified` blocks need to supply an additional `path` key indicating which file modification block to trigger.

  Old syntax

```
trigger:
    - on_startup
    - on_modified:path/to/file
```

New syntax:

```
trigger:
    - block: on_startup
    - block: on_modified
      path: path/to/file
```

- Template metadata is now copied to compilation targets, including permission bits. Thanks to @sshashank124 for the implementation!
- The `trigger` action now follows recursive `trigger` actions. Beware of circular trigger chains!
- `recompile_modified_templates` has been renamed to `reprocess_modified_files`, as this option now also includes copied files.
- Astrality will now only recompile templates that have already been compiled when `reprocess_modified_files` is set to `true`.
- The `template` compile action keyword has now been replaced with `content`. This keyword makes more sense when we add support for compiling all templates within a directory. It also stays consistent with the new action types that have been added.

  Old syntax

```
compile:
    - template: path/to/template
```

  New syntax:

```
compile:
    - content: path/to/template
```

- The module list items within the module `requires` option is now a dictionary, where shell commands are specified under the `shell` keyword. This allows other requirement types (see Added section).

  Old syntax

```
requires:
    - './shell/script.sh'
```

  New syntax:

```
requires:
    - shell: './shell/script'
```

- Astrality now automatically quits if there is no reason for it to continue running.
- When no compilation target is specified for a compile action, Astrality now creates a deterministic file within `$XDG_DATA_HOME/astrality/compilations` to be used as the compilation target. This behaves better than temporary files when programs expect files to still be present after Astrality restarts.
- Astrality is now more conservative when killing duplicate Astrality processes by using a *pidfile* instead of `pgrep -f astrality`.

Fixed

- If a `import_context` action imported specified `from_section` but not `to_section`, the section was not imported at all. This is now fixed by setting `to_section` to the same as `from_section`.
- Template path placeholders are now normalized, which makes it possible to refer to the same template path in different ways, using symlinks and `..` paths.
- Module option `requires_timeout` is now respected.
- Astrality no longer kills processes containing "astrality" in their command line invocation.

1.10 How to contribute

First, thanks for considering contributing to Astrality, that means a lot! Here we describe how you can help out, either by improving the documentation, submitting issues, or creating pull requests.

If you end up contributing, please consider adding yourself to the file `CONTRIBUTORS.rst`.

1.10.1 Bug reports and feature requests

You can browse any existing bug reports and feature requests on Astrality's GitHub issues page on GitHub. New issues issues can be submitted the GitHub create issues page.

1.10.2 Improving the documentation

If you find something you would like to improve in the documentation, follow these steps:

- Navigate to the page that you would like to edit on https://astrality.readthedocs.io.
- Press the "Edit on GitHub" link in the upper right corner.
- Press the "pencil" edit icon to the right of the "History" button.
- Make the changes you intended.
- Write a title and description for your change on the bottom of the page.
- Select the radio button marked as: "Create a new branch for this commit and start a pull request".
- Press "Propose file change".

The documentation is written in the "RestructuredText" markup language. If this is unfamiliar to you, take a look at this RST cheatsheet for more information.

1.10.3 Contributing code

Getting up and running

Cloning the repository

First we need to clone the repository. Open your terminal and navigate to the directory you wish to place project directory and run:

```
git clone https://github.com/jakobgm/astrality
cd astrality
```

Installing python3.6

Astrality runs on `python3.6`, so you need to ensure that you have it installed. If you have no specific preferred way of installing software on your computer, you can download and install it from here. Alternatively, if you use brew on MacOS, you can install it by running:

```
brew install python3
```

Or on ArchLinux:

```
sudo pacman -S python
```

Installing dependencies into a virtual environment

You should create a separate python3.6 "virtual environment" exclusively for Astrality. If this is new to you, take a look at the official virtualenv tutorial.

A quick summation:

```
python3.6 -m venv astrality-env
source astrality-env/bin/activate
```

Your terminal prompt should now show the name of the activated virtual environment, for example `(astrality-env) $ your_commands_here`. You can double check your environment by running `echo $VIRTUAL_ENV`. Later you can deactivate it by running `deactivate` or restarting your terminal. The activated virtual environment is necessary in order to run the developer version of Astrality, including the test suite.

Now you can install all the developer dependencies of Astrality into the virtual environment by running:

```
pip3 install -r requirements.txt
```

You should now make sure that the environment variable `PYTHONPATH` is set to the root directory of the repository. Check it by running:

```
$ echo $PYTHONPATH
/home/jakobgm/dev/astrality
```

With `/home/jakobgm/dev/astrality` being whatever makes sense on your system. If the value is incorrect you should run the following from the repository root:

```
export PYTHONPATH=$(pwd)
```

Running the developer version of Astrality

You should now be able to run the developer version of Astrality by running the following command:

```
./bin/astrality
```

Writing code

The python code in Astrality follows some conventions which we will describe here.

The structure of the code base

A brief outline Astrality's code base is provided in *the API documentation*, and is recommended reading for any new contributor.

Tests

Astrality strives for 100% test coverage, and all new lines of code should preferably be covered by tests. That being said, if testing is unfamiliar to you, submitting code without test coverage is better than no code at all.

Tests are written with the pytest test framework, and you can read a "getting started" tutorial here.

You can run the test suite from the root of the repository by running:

```
pytest
```

Warning: For now, it is important that you run pytest from the root of the repository, else you will get a whole lot of `ModuleNotFoundError` exceptions.

Additionally, there are some tests which are hidden behind the `--runslow` flag, as some tests are slow due to writing files to disk and running certain shell commands. These slow tests can be run by writing:

```
pytest --runslow
```

When you submit a pull request, travis-ci will automatically check if all the tests pass with your submitted code. Coveralls will also check if the test coverage decreases.

If this feels intimidating, do not worry. We are happy to help guide you along if you encounter any issues with testing, so please submit pull requests even if the test suite fails for some reason.

Type annotations

Astrality's code base heavily utilizes the new static type annotations available in python3.6.

The correctness of the type annotations are ensured by using mypy. You can check for type errors by running the following command from the repository root:

```
mypy .
```

`mypy` is a part of the test suite, enabled by the pytest-mypy plugin. Therefore, if the test suite passes, `mypy` must also be satisfied with your code!

All non-testing code should be completely type annotated, as strictly as possible. If this is new to you, or if you want to learn more, I recommend reading mypy documentation.

The offer to help with testing also holds for type annotations of course!

Continuous testing

Although this is mainly a matter of taste, running tests continuously while writing code is a great feedback mechanism.

pytest-watch should be already be installed on your system as part of Astrality's developer dependencies. You can use it to rerun the test suite every time you save any `*.py` file within the repository.

You can run it in a separate terminal by running:

```
ptw
```

It is often useful to run `pytest-watch` in verbose mode, stop on first test failure, and only run one specific test file at a time. You can do all this by running:

```
ptw -- -vv -x astrality/tests/test_compiler.py
```

Debugging

If you end up breaking any behaviour during development, it *should* often be reported by the test suite. Breaking tests will often lead you in the correct direction for fixing the problem.

Some tests might be a bit too brittle, so if you change any underlying data structures it might break some badly written test(s). Sometimes the correct thing to do is to simply delete the failing test. Just ask if you are unsure.

You can also look at the logging output of Astrality in order to pinpoint possible reasons for any weird behaviour. You can set the logging level of astrality by setting the environment variable `ASTRALITY_LOGGING_LEVEL` to an appropriate value, for example:

```
# Set the appropiate logging level
export ASTRALITY_LOGGING_LEVEL=DEBUG

# Run the CLI entrypoint
./bin/astrality
```

If you submit a bug report, we appreciate if you include the standard output of Astrality run with `ASTRALITY_LOGGING_LEVEL=DEBUG`.

Code style

We use the python source code checker flake8 to help us maintain a consistent style across the code base. It runs automatically as part of our `pytest` test-suite.

You can lint your code locally by running `flake8 .` from the root of the repository. Integrating `flake8` into your workflow is recommended, there are plugins available for most popular IDEs and text-editors!

You can instruct git to ensure flake8 compliance before every commit by running `git config --bool flake8.strict true` from your shell.

In addition to this, some additional styling conventions are applied to the project:

- String literals should use single quotes. With other words: `'this is a string'` instead of `"this is a string"`.
- Try to use keyword arguments when calling functions, unless it is extremely clear from context.
- Function arguments split over several lines should use trailing commas. With other words, we prefer to write code like this:

```
compile_template(
    template=template,
    target=target,
)
```

Instead of this:

```
compile_template(
    template=template,
    target=target
)
```

These conventions are mainly enforced in order to stay consistent for choices where `PEP 8` do not tell us what to do.

Local documentation

Astrality uses the sphinx ecosystem in conjunction with readthedocs for its documentation.

You can run a local instance of the documentation by running:

```
cd docs
sphinx-autobuild . _build
```

The entire documentation should now be available on http://127.0.0.1:8000. When you edit the documentation files placed with `docs`, your web browser should automatically refresh the website with the new content!

1.11 API documentation

This section contains documentation for the source code of Astrality, and is intended for the developers of Astrality.

Contents

1.11.1 The structure of the code base

Here we offer a quick overview of the most relevant python modules in the code base, loosely ordered according to their execution order.

`bin.astrality`: The CLI entry point of Astrality, using the standard library `argparse` module.

`astrality.astrality`: The main loop of Astrality, binding everything together. Calls out to the different submodules and handles interruption signals gracefully.

`astrality.config`: Compilation and pre-processing of the user configuration according to the heuristics explained in the documentation.

`astrality.github`: Retrieval of modules defined in GitHub repositories.

astrality.module: Execution of actions defined in modules.

Each module in the user configuration is represented by a `Module` object. All `Module`-objects are managed by a single `ModuleManager` object which iterates over them and executes their actions.

astrality.requirements: Module for checking if module requirements are satisfied.

astrality.actions: Module for executing actions such as "import_context", "compile", "run", and "trigger".

astrality.event_listener: Implements all the types of module event listeners as subclasses of `EventListener`.

astrality.context: Defines a dictionary-like data structure which contains context values, passed off to Jinja2 template compilation.

astrality.compiler: Wrappers around the `Jinja2` library for compiling templates with specific context values.

astrality.filewatcher: Implements a file system watcher which dispatches to event handlers when files are modified on disk.

astrality.utils: Utility functions which are used all over the code base, most importantly a wrapper function for running shell commands.

1.11.2 Modules

Astrality's modules are placed within the `astrality` mother-module. For example, the `actions` module is importable from `astrality.actions`.

Actions module

1.12 Attributions

1.12.1 Inspirations for themes

Themes have been made by the help of several posts on the /r/unixporn subreddit. Here are some of them:

- Default: Still on the lookout for where I got this theme originally
- Tower: Reddit user /u/saors: /r/unixporn post 1.
- Tower: Reddit user /u/TheFawxyOne: /r/unixporn post 2.

In addition, the astrality logo is a modified version of a logo designed by miscellaneous from Flaticon.

1.13 License

Distributed under the terms of the MIT license, Astrality is free and open source software.

```
The MIT License (MIT)

Copyright (c) 2018 Jakob Gerhard Martinussen

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
```

(continues on next page)

(continued from previous page)

```
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```