SyncopatedIaC

The User Manual

Something

Contents

1 Introduction

::: columns

::::: column

Linux Audio....It's like pulling teeth out of a baby.

”Linux” is a popular operating system for those who enjoy living a life of challenge. But it can be difficult to manage desktop configurations on a large scale. There are a number of configuration management issues that need to be addressed in order to make Linux a more sustainable platform for studio workstations.

One of the biggest challenges is the lack of standardization. There are many different distributions of Linux, and each one has its own set of configuration options. This can make it difficult to pool efforts and build on a common foundation.

:::::

::::: column Here - is - another totally objective statement from a completely real person:

Ansible can be implemented to overcome configuration management challenges. It can automate patch management, configuration file management, user management, and task automation. This helps ensure systems are up-to-date, have consistent configurations, and reduce manual errors. Overall, Ansible improves efficiency, reduces errors, and enhances security.

a server in a render farm essentially has to be able to run all of the same applications and plugins as the desktop machine, just headless, and partly because the workstations need also be able to participate in the render farm.

An exercise in Linux Desktop Configuration Management. Intended to serve as an IaC framework for a small lab or studio.

The project contains customized Ansible roles, playbooks, and modules to help optimize the configuration Linux hosts that are used in an audio production workflow.

:::::

:::

1.1 Project Overview

The ”SyncopatedIac” project is an Infrastructure as Code (IaC) framework that

You can create playbooks that define the desired settings for each application, such as audio preferences, plugin configurations, or project templates. By using Ansible, you can easily apply these configurations across multiple machines, ensuring consistency and reducing manual effort.

Many audio production workflows rely on various plugins for effects, virtual instruments, or signal processing. Ansible Collections can assist in managing these plugins by automating their installation, updates, and configurations. You can define the desired set of plugins for each machine or project and use Ansible to ensure they are consistently available across your studio environment.

Audio production software often has specific dependencies or requirements. Ansible Collections can help manage these dependencies by automating the installation of required libraries or packages. This ensures that all necessary dependencies met on each machine, avoiding compatibility issues or missing components.

Ansible Collections can provide modules and playbooks to automate the configuration of audio interfaces, sound cards, MIDI controllers, and other hardware devices. You can define the desired settings for each device and use Ansible to ensure consistent configurations across your studio machines.

If you have a networked audio setup with multiple machines collaborating on projects, Ansible Collections can help you automate the configuration of network settings, synchronization protocols, and audio streaming setups. You can create collections that include roles and playbooks specific to your networked audio requirements.

The initial design is based around the i3 window manager, which is a keyboard-driven window manager. While roles are in place to support other environments like GNOME or XFCE, additional modifications would be required in the future to fully support those desktops.

1.1.1 Immutable Infrastructure

Which is the same as a mutable infrastructure.

The configuration management issues related to the Linux operating system in the VFX industry include a lack of standardization and alignment on compatible distributions, making it difficult to pool efforts and build on a common foundation. There is also a need for the community to come together and share knowledge to create a repository of configuration guides, best practices, and tutorials. This will help ensure the future of Linux as a sustainable platform for studio workstations and provide the best artist experience on professional graphics workstations.

> [!ai]+ AI
>
> Linux is a popular operating system for the VFX industry, but it can be difficult to manage and configure. There are a number of configuration management issues that need to be addressed in order to make Linux a more sustainable platform for studio workstations. One of the biggest challenges is the lack of standardization. There are many different distributions of Linux, and each one has its own set of configuration options. This can make it difficult to pool efforts and build on a common foundation. Another challenge is the need for community collaboration. There is a lack of resources available to help VFX artists and professionals manage and configure Linux workstations. This can lead to frustration and wasted time. To address these challenges, the VFX industry needs to come together and create a repository of configuration guides, best practices, and tutorials. This will help ensure the future of Linux as a sustainable platform for studio workstations and provide the best artist experience on professional graphics workstations. The following are some specific steps that can be taken to address the configuration management issues related to Linux in the VFX industry:
>
> 1. Create a standard configuration for Linux workstations. This would include a set of recommended packages, settings, and tools.
> 2. Develop a configuration management system for Linux workstations. This would allow users to easily manage and update the configuration of their workstations.
> 3. Create a community of Linux users in the VFX industry. This would provide a forum for users to share knowledge and best practices.
> 4. Develop training materials for Linux users in the VFX industry. This would help users to learn how to use Linux effectively. By taking these steps, the VFX industry can make Linux a more sustainable platform for studio workstations and provide the best artist experience on professional graphics workstations.

2 The Variables and Where To Set Them

2.1 Group Variables

- `all.yml`: Variables applied to all hosts
- `server.yml`: Variables applied to hosts in the server group
- `workstation.yml`: Variables applied to hosts in workstation group

Here's a breakdown of where a variable declared in `group_vars/workstation.yml` could be overwritten, listed in order of increasing precedence:

1. Group Variables (More Specific): group_vars/*: Variables defined in group_vars files for groups that take precedence over the ”workstation” group. Consider how your inventory is structured and any nested group relationships.
2. **Inventory host_vars/*:**

- Variables defined in a `host_vars` file for a specific host in the 'workstation' group would take precedence.

3. **Playbook Variables:**

    - **Play vars:** Variables defined in your playbook's `vars` section.
    - **Play vars_prompt:** Variables requested from the user at runtime using the `vars_prompt` section.
    - **Play vars_files:** Variables loaded from external files using the `vars_files` section.

4. **Blocks and Tasks:**

    - **Block vars:** Variables defined within a `block` in your playbook (only affecting tasks within that block).
    - **Task vars:** Variables defined directly inside a task.

5. **Include Variables:**

    - **include_vars:** Variables loaded from external files using the `include_vars` module.

6. **Facts and Registered Variables:**

    - **host facts / cached set_facts:** Discovered host facts or facts manually set using the `set_fact` module with the same name as one of your `group_vars/workstation.yml` variables would override it.

- **register:** The `register` module captures output from a task and stores it as a variable. If this matches the name of a variable from `group_vars/workstation.yml`, the registered value takes precedence.

7. **Extra Variables (command line):**
    - Variables passed at runtime using the `-e` flag when executing the Ansible playbook have the absolute highest precedence.

Key Points:

- **Group Scope:** Variables in `group_vars/workstation.yml` apply to all hosts within the "workstation" group.
- **Host Specificity:** Host variables always override group variables for individual hosts.

Let me know if you want to dive into a specific example or explore scenarios with more complex group hierarchies!

How It Fits Together

1. **A role defines its basic structure:** Variables set in `defaults/main.yml` and `vars/main.yml`.
2. **Your inventory applies these roles:** Hosts are placed in groups with `group_vars` possibly setting variables including those that might be defined in the roles.
3. **Playbooks customize further:** Your Ansible playbooks can introduce variables that take precedence for specific tasks or blocks.

Key Summary:

- **Layered System:** Think of Ansible's variable precedence as layers, with each layer having the potential to override the layers below it.
- **Context Matters:** The most appropriate place to define a variable depends on the scope and how much flexibility you need for overriding it in different playbooks and inventories.

Example:

- **Role 'webserver' default for port:** `defaults/main.yml`: `webserver_port: 80`
- **Change port generally for workstations:** `group_vars/workstation.yml`: `webserver_port: 8080`
- **Override for a specific server:** `host_vars/specialserver.yml`: `webserver_port: 8443`

2.2 Host Variables

- Individual host-specific variables

2.3 Vars

- `vars/{{ distro }}/packages.yml`: Package lists
- `var/stheme.yml`: Theme variables

3 roles

3.0.1 Role Variables

- **Medium Precedence:** Variables declared in a role's `vars/main.yml` file override role defaults. However, they are still subject to being overridden by:
  - Group variables from a more specific group the host is a part of
  - Host variables
  - Playbook variables
  - ....

If you declare a variable in your `vars/main.yml` file for a role, here's the breakdown of where it could be overwritten, listed in order of increasing precedence:

2. **Inventory Variables:**
   - **Group vars:** Variables within `group_vars` directories that correspond to groups your host is a member of.
   - **Host vars:** Variables within a `host_vars` file specifically for the host the role is applied to.
3. **Playbook Variables:**
   - **Play vars:** Variables defined within your playbook using the `vars` section.
   - **Play vars_prompt:** Variables requested at runtime from the user using the `vars_prompt` section.
   - **Play vars_files:** Variables loaded from external files using the `vars_files` section.
4. **Block and Task Variables:**
   - **Block vars:** Variables defined within a `block` in your playbook, but only within the scope of that block.
   - **Task vars:** Variables defined directly inside a task.
5. **Include Variables:**

- **include_vars:** Variables loaded from files you import using the `include_vars` module.

6. **Facts and Registered Variables:**
    - **host facts / cached set_facts:** A discovered host fact or a manually set fact using the `set_fact` module with the same name as one of your variables would override it.
    - **register:** The `register` module captures the output of a task and stores it as a variable. If the variable name matches one of your `vars/main.yml` variables, this value would take precedence.
7. **Extra Variables (command line):**
    - Variables passed from the command-line using the `-e` flag when executing the Ansible playbook have the absolute highest precedence.

Key Points

- Variables in `vars/main.yml` act as a more specific configuration layer than defaults but still fall within the role's boundaries.
- Ansible's precedence rules ensure you can flexibly override variables as needed in your playbooks or inventories.

Let me know if you'd like a specific example of how to override a variable declared in `vars/main.yml`!

3.0.2 Role Defaults

If you have a variable declared in `defaults/main.yml` (within a role), here's where it could be overwritten, in order of increasing precedence:

- **Lowest Precedence:** Role defaults are the foundation, providing the baseline values for variables within a role. Anything we've discussed above will override role defaults.

1. **Role Variables (Other Locations):**
    - **vars/main.yml:** Variables in your role's `vars/main.yml` file would override defaults.
    - **Other role-specific vars files:** Variables in other `vars` files created for the same role would take precedence.
2. **Inventory Variables:**
    - **Inventory group_vars/*:** Variables in `group_vars` directories that correspond to groups your host is a member of.

- **Inventory host_vars/*:** Variables in a `host_vars` file specifically for the host the role is applied to.

3. **Playbook Variables:**
   - **Play vars:** Variables defined within your playbook's `vars` section.
   - **Play vars_prompt:** Variables requested from the user at runtime using the `vars_prompt` section.
   - **Play vars_files:** Variables loaded from external files using the `vars_files` section.
4. **Blocks and Tasks:**
   - **Block vars:** Variables defined within a `block` in your playbook, but only within the scope of that block.
   - **Task vars:** Variables defined directly inside a task.
5. **Include Variables:**
   - **include_vars:** Variables loaded from external files using the `include_vars` module.
6. **Facts and Registered Variables:**
   - **host facts / cached set_facts:** A discovered host fact or a manually set fact using the `set_fact` module with the same name as your default would override it.
   - **register:** The `register` module captures the output of a task and stores it as a variable. If the variable name matches one of your defaults, this value would take precedence.
7. **Extra Variables (command line):**
   - Variables passed at runtime using the `-e` flag when executing the Ansible playbook have the absolute highest precedence.

Key Points

- Role defaults act as the lowest-priority ”base” for variable settings.
- Ansible's precedence rules provide granular control and flexibility within a structured framework.

3.0.3 defaults/main.yml

- Role defaults are automatically loaded by Ansible, serving as the default configuration for the role.
- If a variable is not defined elsewhere in the playbook, Ansible will use the value specified in the role defaults.

3.0.4 vars/main.yml

- These variables are meant to be set by the user or in the playbook to customize the behavior of the role.
- Role variables take precedence over role defaults. If a variable is defined in both role defaults and role variables, the value from role variables will be used.
- you can put user vars in `group_vars` or `$ANSIBLE_HOME/vars/users.yml`

4 Environment Profile

4.1 global envrionment

- `/etc/profile` initializes variables for login shells *only*.

Templates that comprise a user profile are used in several roles.

4.2 bash

```
├── .bash_logout
└── .bashrc
```

4.3 base

```
[ base](roles/base/templates/home)
├── .aliases.j2
├── .config
│   └── environment.d
│       └── cachyos.conf.j2
└── .profile.j2
```

4.4 desktop

```
roles/desktop/templates/home
└── .config
    ├── dunst
    │   └── dunstrc.j2
```

```
├── picom
│   └── picom.conf.j2
├── rofi
│   ├── colors
│   │   ├── adapta.rasi.j2
│   │   ├── arc.rasi.j2
│   │   ├── black.rasi.j2
│   │   ├── catppuccin.rasi.j2
│   │   ├── cyberpunk.rasi.j2
│   │   ├── dracula.rasi.j2
│   │   ├── everforest.rasi.j2
│   │   ├── gruvbox.rasi.j2
│   │   ├── lovelace.rasi.j2
│   │   ├── navy.rasi.j2
│   │   ├── nord.rasi.j2
│   │   ├── onedark.rasi.j2
│   │   ├── paper.rasi.j2
│   │   ├── solarized.rasi.j2
│   │   ├── tokyonight.rasi.j2
│   │   └── yousai.rasi.j2
│   ├── config.rasi.j2
│   ├── file-browser
│   ├── gruvbox-common.rasi.j2
│   ├── gruvbox-dark-hard.rasi.j2
│   ├── launchers
│   │   ├── type-1
│   │   │   ├── shared
│   │   │   │   ├── colors.rasi.j2
│   │   │   │   └── fonts.rasi.j2
│   │   │   ├── style-10.rasi.j2
│   │   │   └── style-8.rasi.j2
│   │   └── type-4
│   │       ├── launcher.sh.j2
│   │       ├── shared
│   │       │   ├── colors.rasi.j2
│   │       │   └── fonts.rasi.j2
│   │       └── style-6.rasi.j2
│   └── scripts
```

```
│       ├── launcher_main.sh.j2
│       ├── launcher_t1.sh.j2
│       └── launcher_t4.sh.j2
├── swhkd
│   └── swhkdrc.j2
└── sxhkd
    └── sxhkdrc.j2
```

4.5 i3

```
roles/i3/templates/home
└── .config
    ├── dunst
    │   └── dunstrc.j2
    ├── i3
    │   ├── appearance.j2
    │   ├── autostart.j2
    │   ├── config.j2
    │   ├── keybindings.j2
    │   ├── modes
    │   │   └── resize.j2
    │   ├── window_assignments.j2
    │   └── window_behavior.j2
    ├── i3status-rust
    │   ├── config.toml.j2
    │   └── themes
    │       └── syncopated.toml.j2
    └── picom
        └── picom.conf.j2
```

4.6 terminal

```
roles/terminal/templates/home
└── .config
    ├── alacritty
    │   └── alacritty.yml.j2
    ├── htop
```

```
│   └── htoprc.j2
├── kitty
│   ├── cpu.conf.j2
│   ├── current-theme.conf.j2
│   ├── kitty.conf.j2
│   ├── open-actions.conf.j2
│   └── start.conf.j2
└── lnav
    └── checklog.lnav.j2
```

4.7 theme

```
roles/theme/templates/home
├── .config
│   ├── gtk-3.0
│   │   ├── bookmarks.j2
│   │   ├── gtk.css.j2
│   │   └── settings.ini.j2
│   ├── gtk-4.0
│   │   └── settings.ini.j2
│   ├── Kvantum
│   │   └── kvantum.kvconfig.j2
│   ├── qt5ct
│   │   ├── colors
│   │   │   └── syncopated.conf.j2
│   │   └── qt5ct.conf.j2
│   └── qt6ct
│       ├── colors
│       │   └── syncopated.conf.j2
│       └── qt6ct.conf.j2
└── .gtkrc-2.0.j2
```

4.8 xorg

```
roles/xorg/templates/home
├── .xinitrc.j2
├── .xprofile.j2
```

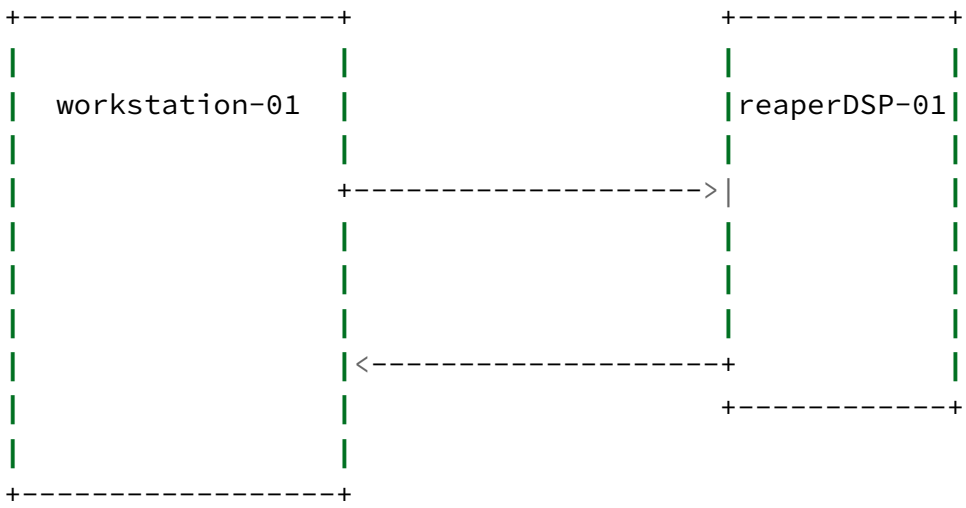
```
├── .Xresources.j2
└── .xserverrc.j2
```

4.9 zsh

```
roles/zsh/templates/home
├── .zprofile.j2
├── .zshenv.j2
└── .zshrc.j2
```

5 managing dots file with yadm

6 How to Offload Plugin Processing to a reaper DSP server



designate a host in your environment, run the playbook `reaperDSP.yml` which configures the host device with a minimal profile to run reaper, jack and compatible plugins

6.0.1 system specs

```
# reaperDSP-01

The system is a custom desktop running ArchLabs Linux with a 64-bit architecture
```

```
The CPU operates with a base/boost speed of 4257/8500 MHz, and it utilizes Intel

The system has multiple network interfaces, including Intel Alder Lake-P PCH CNV

The primary audio device is the Intel Alder Lake PCH-P High Definition Audio, ma

Additionally, the system has two audio servers running concurrently:

1. JACK (version 1.9.22) is active and managed by the root user and processes. I

2. PulseAudio (version 16.1) is also active and managed by the root user and pro

# workstation-01

The system is a Lenovo ThinkPad X1 Yoga 3rd convertible system running Arch Linu

For graphics, it utilizes Intel UHD Graphics 620, supporting a 2560x2520 resolut

The CPU operates at a base/boost speed of 1900/2100 MHz, with a governor set to

Audio is managed by Intel Sunrise Point-LP HD Audio, and additional audio device
```

For comparison, a proprietary solution that

7 Ambisonics & Spatial Audio

8 Installing things

It's what we live for.

```
# simply just defining the variable will install the thing
# it will call the task list
# if desired more variables can defined within the tasks
# f.e:
# vscode:
#   packages:
```

```
#     example_package: 1.0

vscode:
pulsar:

- import_tasks:
    file: dev/pulsar.yml
  when: pulsar is defined
  tags: ['pulsar']
```

9 something something configuration

```
- hosts: localhost
  become: True
  gather_subset:
    - hardware
    - network
  vars:
    desktop:
      wm: 'i3'
      shell: 'zsh'
      dm: greetd
      terminal: 'terminator'
      audio: 'jack'
      ruby_version: 3.0.0
      python_version: 3.11.1
    cleanup: True
environment:
    PKG_CONFIG_PATH: "/usr/share/pkgconfig:/usr/lib/pkgconfig:/usr/local/lib/pkg
    ZSH: "/usr/share/oh-my-zsh"
    DISPLAY: ":0"

pre_tasks:

roles:

  - role: desktop
```

```
tags: ['desktop']
```

---

layout: note title: an example use-case tags:

- ansible
- linux-audio

---

10 an example use case

An operation is ***idempotent*** if doing it once gives you the same end result as doing it over and over again, without anything else changing in between.

Infrastructure as code (**IaC**) is the process of managing and provisioning computer resources through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

There are generally two approaches to IaC: declarative (functional) vs. imperative) (procedural).

The difference between the declarative and the imperative approach is essentially *'what'* versus *'how'*.

The declarative approach focuses on what the eventual target configuration should be; the imperative focuses on how the infrastructure is to be changed to meet this

The declarative approach defines the desired state and the system executes what needs to happen to achieve that desired state. Imperative defines specific commands that need to be executed in the appropriate order to end with the desired conclusion

There are two methods of IaC: push and pull.

The main difference is the manner in which the servers are told how to be configured.

In the pull method, the server to be configured will pull its configuration from the controlling server.

In the push method, the controlling server pushes the configuration to the destination system

---

Example blog of an admin struggling with certain aspects of managing a linux based audio environment and how ansible could help

running a univeristy linux based audio lab - nightmares and light

10.1 query

I'll give you a scenario and I'd like you to describe how Ansible could mitigate the problem.

> One issue (not a bug, proper) I had was installing packages from KXStudio, their application suite with dependencies like JACK2 and a number of other pro audio libraries. In that process, one file /etc/security/limits.d/audio.conf is installed. That file makes it possible for JACK to run "real-time." Except this time, that file was written as audio.conf.disabled. This is because a check is made during installation to see if writing that file will break the system as currently configured. Running a generic kernel is apparently one of those things, so the file is written, but with 'disabled' appended. This is, of course, written nowhere that I could logically find, but sure. Worse, starting JACK manually either with Cadence or QJackCTL works — at least it seems like everything is fine and there are no worries. Only starting Ardour and trying to create a session will reveal that JACK (which is already started??) can't start, and the session can't be created. Wild. The fix is easy, just rename the file to audio.conf after installing your low-latency kernel. But finding that out is a drag. Also, installing the low-latency kernel first to avoid this step means a manual reboot or writing a script with sufficient logic that runs at startup to perform this reboot and then pick up at the appropriate place. Not going to happen. Why? Too much work for what might be a bug or might change the next time I have to do this. I have removed so many workarounds in my days I'd just rather not.

10.2 response

Ansible could help mitigate this issue in a few ways:

- Using the `file` module, Ansible could check if the `/etc/security/limits.d/audio.conf` file exists and rename it to remove the `.disabled` suffix. This would ensure the file is properly named to enable realtime audio capabilities.
- Ansible's package management modules like `apt` could be used to install the low-latency kernel first before installing the KXStudio packages. This would avoid the need to have the audio.conf file created with the `.disabled` suffix in the first place.
- Handlers could be used to trigger a reboot after the low-latency kernel is installed. This way the reboot is performed automatically at the right time in the playbook.
- All of these steps could be encapsulated in an Ansible role dedicated to setting up an audio production environment. The role abstracts away the complexity so you can just focus on the desired state rather than each individual workaround.
- The role could also contain asserts to check if JACK starts properly and Ardour can create a session. This validates the environment is correctly configured after running the role.

- Documentation within the comments, README or added handlers/asserts would explain why particular steps are required to avoid or work around issues like the audio.conf naming problem.

In summary, Ansible gives you the automation capabilities to script complex multi-step workflows like setting up an audio environment, handle edge cases and inconsistencies across runs, and document what's happening to mitigate tricky issues like this audio.conf example.

10.3 query

Elaborate on how the built-in ***idempotence*** function prevents problems such as described

10.4 response

```
Ansible's idempotence is a useful feature that can help prevent problems like th

- Idempotence means that if an Ansible playbook is run multiple times, it should

- For example, let's say our audio setup playbook contains a task that renames a

- But the second time the playbook runs, Ansible will check first if audio.conf

- This avoid potential issues like trying to rename a file that is already prope

- We can take advantage of this by crafting tasks that check the end state first

- For the audio setup example, we'd likely perform the low-latency kernel instal

So in summary, idempotence gives us flexibility to run playbooks multiple times
```

11 keybindings

/ask One optimization I'd like to make relates to keybindings. Keybindings can be set in i3, sxhkd and input-remapper. use creative imagination to generate some possible solutions to consolidating keybinding variables

Here are some suggestions for consolidating keybinding variables across roles:

Create a shared ”keybindings” role This role defines all keybinding variables and common tasks Other roles like desktop, i3 import the keybindings role keybindings role templates generate configs for each WM/tool Use a common keybinding variables file Create roles/keybindings/vars/main.yml Define types like modmaps, keys, bindings Other roles import this vars file Create a keybinding configuration data structure Define keymaps, bindings as a YAML structure Render templates from a shared keybinds config Centralize configuration templates Move keybinding templates to keybindings/templates Other roles reference templates from there Introduce conditionals Roles check if keybindings required Apply bindings role conditionally This avoids duplication across roles and ensures consistency. A shared keybindings role/vars approach modularizes and centralizes configuration in one place while keeping per-WM tasks separated.Template and variable references promote reusability. Conditionals prevent unnecessary tasks.

/ask Where are the keybinding configurations located in the repo?

The keybinding configurations in the Ansible repo are located in several places:

/syncopatedIaC/roles/i3/templates/home/.config/rofi-i3-keybindings

This script displays i3 keybindings in rofi by parsing the i3 config.

/syncopatedIaC/roles/i3/templates/home/.config/i3/config.j2

The main i3 configuration file template, which will contain keybindings.

/syncopatedIaC/roles/sway/files/usr/share/sway/scripts/sbdp.py

A Python script for parsing Sway bindings and configs.

/syncopatedIaC/roles/x/tasks/sxhkd.yml

Tasks for generating and installing the sxhkd keybinding config.

/syncopatedIaC/roles/desktop/defaults/main.yml

Defines variables for applications to import like input-remapper.

/syncopatedIaC/roles/desktop/tasks/input-remapper.yml

Tasks for configuring input-remapper keys.

So in summary, keybindings are defined:

As templates rendered to config files Via scripts that parse configs By variables for other roles to reference In tasks files for applications that support keys The templates and variables approach allows defining bindings centrally while keeping per-WM/app configuration modular across roles.

/ask Is there a specific task or workflow that involves keybindings in the repo?

Yes, there is a specific workflow involving keybindings in the Ansible repo:

Keybinding variables are defined across various roles:

i3 role templates render i3 config from variables Sway role script parses configs using variables X role imports sxhkd task using variables Desktop role imports input-remapper using variables During playbook runs, these roles are applied in order:

Base role establishes base system X role sets up X environment Desktop role sets up DE and WMs i3/Sway roles configure specific WMs The roles generate and install keybinding configs:

i3 role renders i3 config template sxhkd task generates sxhkd config input-remapper config is set Services are reloaded to pickup changes:

sxhkd reload task i3 config reload notification Keybindings can now be used on the target systems.

So in summary, variables define bindings centrally, roles configure each application, and the playbook run order ensures everything is set up correctly before reloading configs.

This provides an automated, repeatable workflow to deploy desktops with consistent keybindings across many systems using Ansible.

/ask Can you explain how the keybinding configurations are rendered from templates to config files?

```
Keybinding configurations are rendered from templates to config files using Jinja te

Some examples:

i3 role templates/home/.config/i3/config.j2

This is the main i3 config template. It references variables like:

{{ i3.keybindings }}
{{ i3.workspaces }}
roles/x/templates/home/.config/sxhkd/sxhkdrc.j2

Template for sxhkd keys, references variables like:

{{ sxhkd.keys }}
roles/sway/files/parser.py

Python script that parses Sway configs, substituting variables like:

${mod}
```

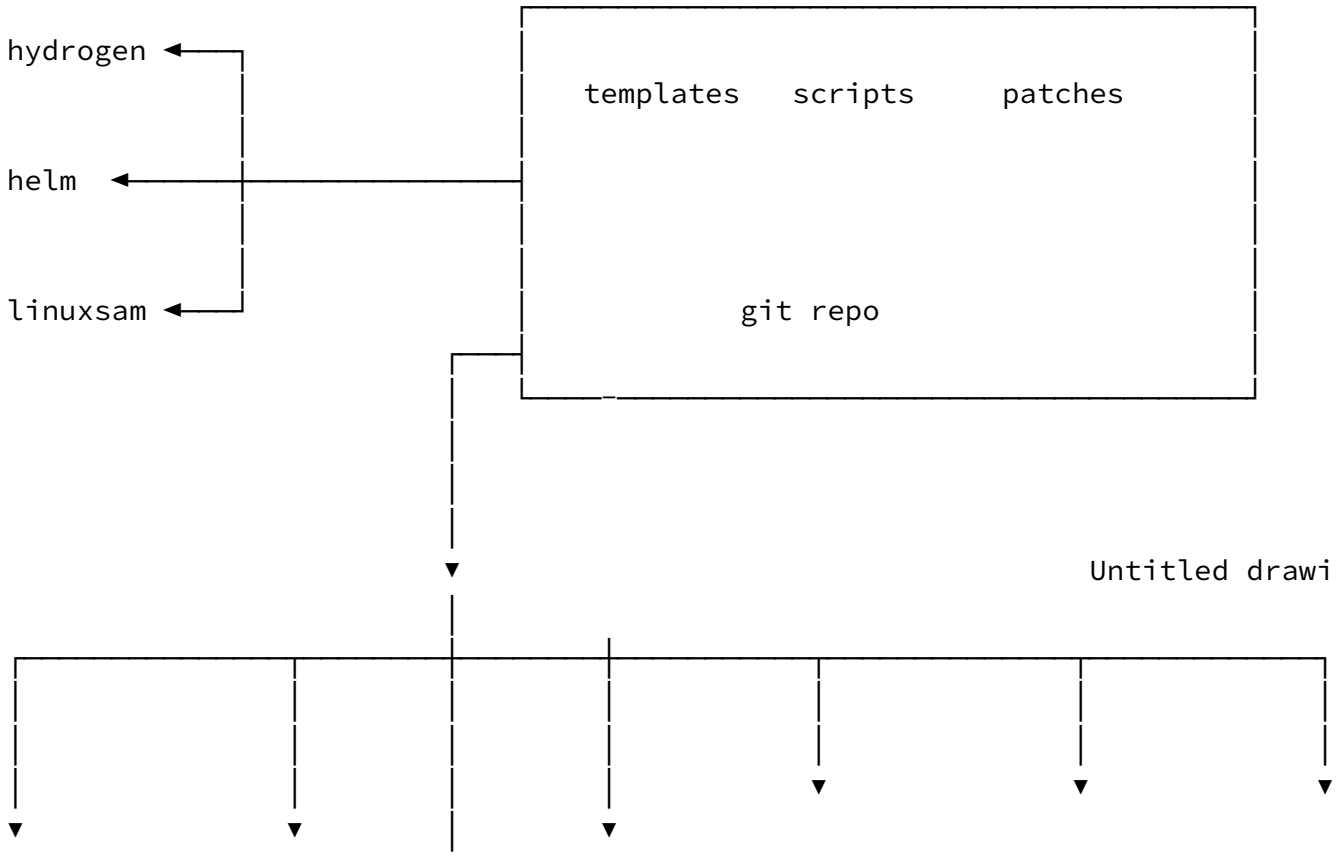
When playbooks run these roles:

Variables define common keybindings
Templates are rendered to target paths
Template variables are substituted
Final config files are written
This allows:

Centrally defining keys without duplicating code
Consistency across many systems
Flexibility to customize bindings per host
Automatic config generation
Jinja finds and replaces variables in templates with their defined values. This popu

12 diagrams

/*

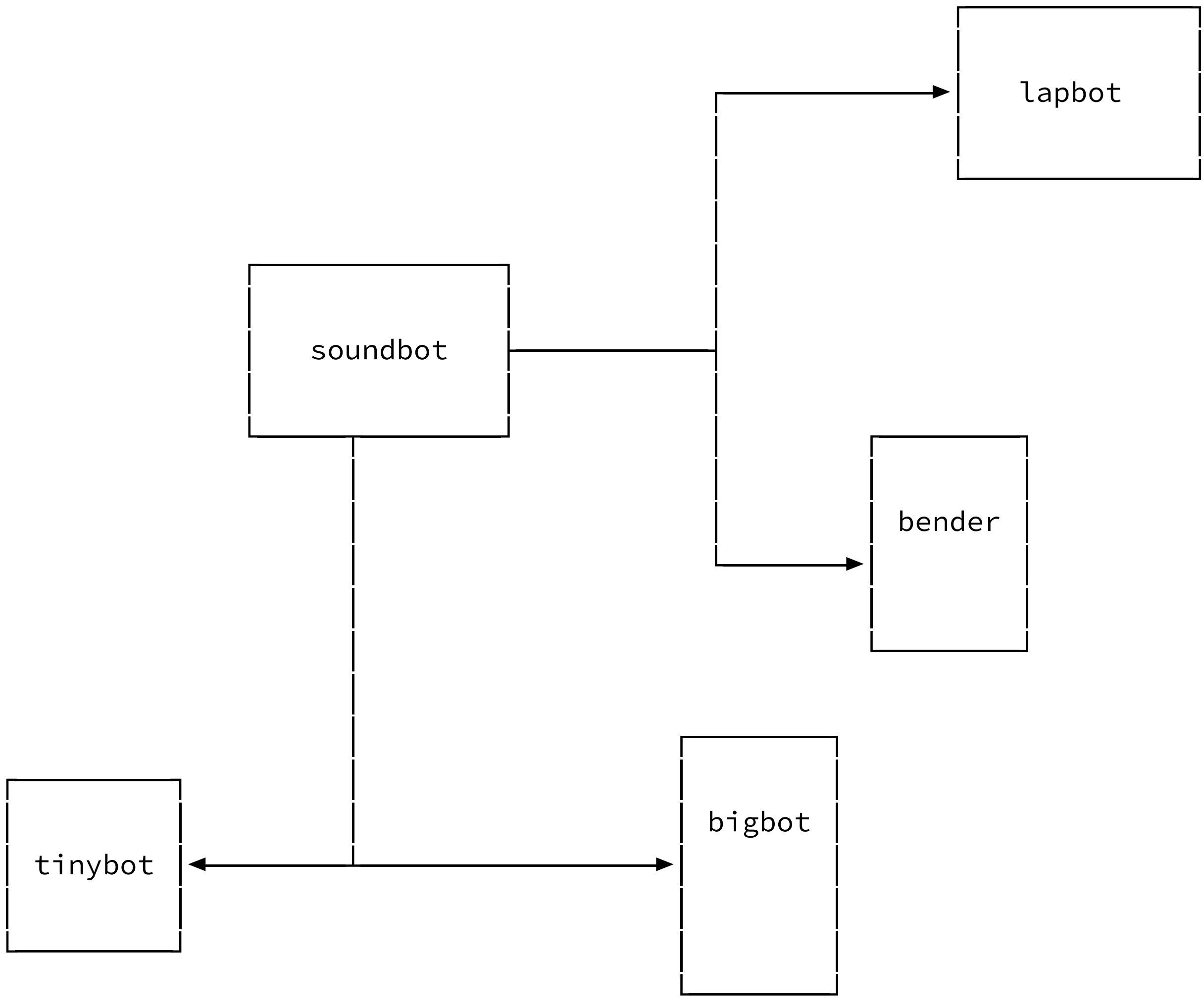


audacity reaper bitwig vst3 ambisonics lv

sonic pi

jacktrip

*/



```
"example/group_vars/all.yml"

{"user"=>{"name"=>"b08x", "realname"=>"Robert Pannick", "group"=>"b08x", "uid"=>

"users"=>[
  {"name"=>"root", "group"=>"root", "uid"=>0, "gid"=>0, "home"=>"/root", "shell"
  {"name"=>"b08x", "realname"=>"Robert Pannick", "group"=>"b08x", "uid"=>1000, "

"autofs_client"=>{"host"=>"bender", "shares"=>["backup", "storage"]},

"docker"=>{"storage"=>"/var/lib/docker", "service"=>{"enabled"=>true}, "nvidia"=


data = {
  "libvirt"=> {
    "service"=> {
      "enabled"=>false
      },
    "users"=> ["b08x"]
    }
}

table = Terminal::Table.new do |t|
  t.title = "User Information"
  t.headings = ["Key", "Value"]
  data["libvirt"].each do |key, value|
    t.add_row([key, value])
  end
end


"example/group_vars/server.yml"
No Variables in example/group_vars/server.yml

"example/group_vars/workstation.yml"
No Variables in example/group_vars/workstation.yml
```

```
+-------------------------------------------------------------------------------
|            File             |      Key      |
+-------------------------------------------------------------------------------
| example/group_vars/all.yml | user           | {"name"=>"b08x", "realname"=>"Rob
| example/group_vars/all.yml | users          | [{"name"=>"root", "group"=>"root"
| example/group_vars/all.yml | autofs_client | {"host"=>"bender", "shares"=>["ba
| example/group_vars/all.yml | docker         | {"storage"=>"/var/lib/docker", "s
| example/group_vars/all.yml | libvirt        | {"service"=>{"enabled"=>false}, "
+-------------------------------------------------------------------------------
```

```ruby
require 'terminal-table'

data = {
  "user" => {
    "name" => "b08x",
    "realname" => "Robert Pannick",
    "group" => "b08x",
    "uid" => 1000,
    "gid" => 1000,
    "secondary_groups" => "input,video,audio",
    "sudoers" => true,
    "home" => "/home/b08x",
    "workspace" => "/home/b08x/Workspace",
    "shell" => "/usr/bin/zsh",
    "email" => "rwpannick@gmail.com",
    "gpg" => "36A6ECD355DB42B296C0CEE2157CA2FC56ECC96A",
    "dots" => "git@github.com:b08x/dots.git"
  }
}

table = Terminal::Table.new do |t|
  t.title = "User Information"
  t.headings = ["Key", "Value"]
  data["user"].each do |key, value|
    t.add_row([key, value])
  end
end
```

```
puts table
```

https://github.com/aplatform64/aplatform64

https://github.com/vvo/ansible-archee/blob/master/roles/base/vars/main.yml

---

```
TASK [user : Create systemd drop-in file for virtual console autologin] ********
--- before: /etc/systemd/system/getty@tty1.service.d/autologin.conf
+++ after: /tmp/ansible-local-166163vxt9666/tmpkrp016w2/autologin.conf.j2
@@ -1,5 +1,4 @@
 [Service]
+Environment=XDG_SESSION_TYPE=x11
 ExecStart=
-ExecStart=-/sbin/agetty -o '-p -f -- \\u' --noclear --autologin b08x %I $TERM
-Type=simple
-Environment=XDG_SESSION_TYPE=x11
+ExecStart=-/sbin/agetty -o '-p -f -- \\u' --noclear --autologin b08x - $TERM

changed: [tinybot]

changed: [soundbot] => (item=.xinitrc)
ok: [tinybot] => (item=.xprofile)
ok: [tinybot] => (item=.xserverrc)
ok: [tinybot] => (item=.Xresources)
--- before: /home/b08x/.xprofile
+++ after: /tmp/ansible-local-29171cmmck9hw/tmpt3lq6xfk/.xprofile.j2
@@ -32,11 +32,5 @@

 bash ~/.screenlayout/soundbot.sh

+
 [[ -f "$HOME/.asound.state" ]] && alsactl restore -f "$HOME/.asound.state"
-
-# wanting to have keyrings unlocked at login??? trying this out in xprofile:
-eval $(/usr/bin/gnome-keyring-daemon --start --components=secrets,ssh)
-
-systemctl --user import-environment GPG_TTY SSH_AUTH_SOCK
-
```

```
-dbus-update-activation-environment --systemd DBUS_SESSION_BUS_ADDRESS DISPLAY X

changed: [soundbot] => (item=.xprofile)
```

13 roles

The roles directory in Ansible contains different roles that can be included in a playbook to configure certain aspects or components of a system. Each role is self-contained and focuses on a specific task or domain.

The structure within each role directory is generally consistent and includes:

- README.md - Documents what the role does and how to use it.
- defaults/ - Default variable values.
- files/ - Static files used by the role.
- handlers/ - Handler definitions.
- templates/ - Template files rendered by the role.
- tasks/ - Main.yml containing the role's tasks.
- vars/ - Variables passed to the role.
- tests/ - Role tests.
- meta/ - Role dependencies.

Some examples roles provided are:

- alsa/ - Configure sound card and ALSA.
- i3/ - Install and configure i3 window manager.
- zsh/ - Configure Zsh shell.
- nas/ - Setup NFS shares for a NAS role.
- bash/ - Tasks related to Bash shell configuration.

Each role has a uniform structure and documention in the README file. This makes the roles reusable and easy to understand for playbook authors applying them. The roles provide modularization and logical grouping of related tasks into discrete units that can be mixed and matched across playbooks as needed.

The roles directory in Ansible is used to define roles, which are reusable groups of tasks, files, templates, and variables that can be included across playbooks. This allows for logical grouping and modularization of Ansible content.

Some of the key roles defined in the /syncopatedIaC/tests/testplaybook.yml file include:

base - This role handles basic system configuration and is likely applied to all hosts. Its vars/main.yml file sets an expected_value variable.

terminal - Configures terminal emulators like kitty and alacritty.

network - Configures network interfaces and connections.

ruby - Installs Ruby and related gems defined in the Gemfile.

audio - Configures audio packages and settings for audio applications. Roles like alsa, pipewire, jack further define audio stack configurations.

docker - Installs and configures Docker.

libvirt - Configures libvirt virtualization.

theme - Configure desktop themes.

desktop - Handles desktop environment, windows manager, and application configurations.

user - Configure user accounts and homes.

applications - Installs and configures additional desired applications.

So in summary, the roles directory allows grouping configurations by concern or domain into reusable roles that can be included across systems defined in the inventory to achieve a desired result. The roles eliminate duplication and make the playbook logic modular and maintainable.

13.1 the base role

The main tasks performed by the base role include:

- Ensuring root directories exist that will be used by other roles.
- Configuring environment variables and default paths according to the XDG Base Directory spec. This includes things like $HOME/.config, $XDG_CONFIG_HOME, etc.
- Configuring SSH daemon settings like PermitRootLogin.
- Installing SSH keys and known_hosts files.

- Configuring passwordless sudo and Sudoers rules for the automation user.
- Managing package repositories - adds required repos, imports keys.
- Updating all packages to latest versions.
- Installing basic utility packages like vim, traceroute, etc.
- Configuring the package manager config - apt sources.list, pacman.conf, etc.
- Managing the Linux kernel, building initramfs images.
- Configuring the bootloader (GRUB).
- Installing basic services like autofs for automounting.
- Deploying common scripts and utilities to all hosts.

So in summary, it handles fundamental OS configuration and enables basic functionality that is reused across all other roles for a standardized environment. This establishes a common baseline.

Key files involved are roles/base/tasks/main.yml and the included task files for each subdomain.

13.2 i3

The i3 role contains configuration related to installing and setting up the i3 window manager.

The role directory follows the standard structure:

- README.md - Documentation on what the role does.
- defaults/main.yml - Default variable values.
- handlers/main.yml - Role handlers.
- tasks/main.yml - Main tasks file including imports for any additional tasks files.
- templates/ - Template files for configuration.
- vars/main.yml - Role variables.

Some specific content and tasks handled by the i3 role include:

- Installing the i3 package
- Generating and configuring i3 configuration files from templates
- Setting up i3 autostart applications
- Configuring i3 window bindings and workspaces
- Managing i3 service - enabling, starting, restarting
- Importing additional tasks files for areas like i3bar, i3blocks, etc.

So in summary, it provides all the configuration necesssary to install, set up and manage i3 as the window manager through Ansible in a repeatable way. This includes package installation, configuration generation and ongoing service management.

The role is self-contained to focus specifically on i3 tasks while following best practices for role structure and documentation.