

A vertical decorative graphic on the left side of the slide, rendered in various shades of red. It features a collage of icons representing technology and automation: a cloud with a keyhole, a database cylinder, a server rack, a terminal window, a person silhouette, and various arrows and symbols like 'X' and 'O' connected by lines, suggesting a workflow or process.

Converting Bash scripts to Playbooks

Automate all the things

David Glaser

Senior Technical Account Manager





The Automation Journey often involves converting scripts to Ansible Playbooks.

This is a perfect time to reexamine how these scripts work and utilize Ansible idempotent features.

Why Convert?

The Shortcut Trap

Ansible includes **shell**, **command**, and **script** modules. These allow for direct running of scripts on remote hosts. This presents problems:

- ▶ No idempotency checking
- ▶ Output is simply the script output
- ▶ Does not support `check_mode`

Conversion Misnomers

What they don't tell you

- ▶ Converting will likely not be 1 for 1
- ▶ There may not be an ansible module for everything
- ▶ The ansible playbook will likely be longer than the bash script, but easier to read
- ▶ The flow of the script may have to be reworked

Bash vs Ansible

Features

Bash is a great scripting language, but all redundancy, idempotency, and error checking must be done manually

- ▶ Multiple functions in each task(line)
- ▶ Non-linear flow (functions) through script
- ▶ Learning curve for syntax and formatting

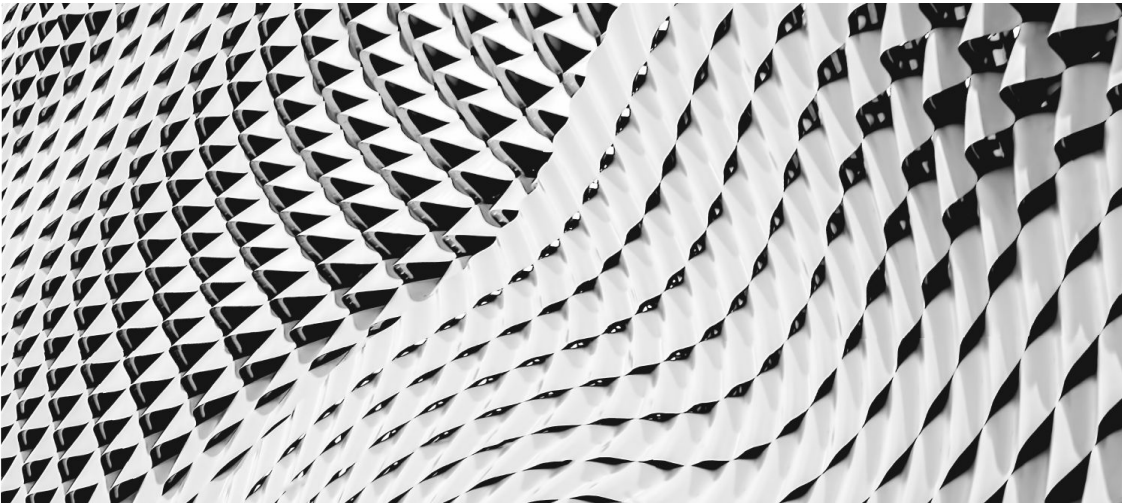
Ansible handles redundancy, idempotency, and error checking when using supported modules

- ▶ One function per task*
- ▶ Linear flow through playbook
- ▶ Learning curve for formatting only**

* Some tasks perform multiple functions, but these are limited to permission changes to the working file, etc.

** Ok, there's some syntax learning, but it's a on module by module basis

Converting Concepts



We'll focus on a method to follow to convert bash scripts to Ansible Playbooks

- ▶ Temporary script file
- ▶ Examine script flow
- ▶ Converting conditionals
- ▶ Examine system commands and arguments
- ▶ Verify functionality

Temporary script file

Much of the planning for conversion means moving code around. Using a temporary file will assure that any changes can be tested or backed out.

Examine Script Flow

Bash has functions, but Ansible does not. It's possible to include groups of tasks using **include** and **import** modules however

- ▶ Look over the Script Flow, are functions used?
 - If so, are they used multiple times?

Used Once

Place the function in line in the script where it is called

Used Multiple Times

- ▶ These lines(tasks) will be in their own file and included or imported into the main playbook
- ▶ Any variables that are 'sent' to the function need to be registered in the playbook.

Examine Script Flow

```
print_function () {  
    echo "Hello World!"  
}  
  
echo "Before function"  
  
print_function  
  
echo "After function"  
  
-----  
  
echo "Before function"  
  
echo "Hello World!"  
  
echo "After function"█
```

Writing the playbook

Rewriting the script into a linear flow will make converting to a playbook easier

- ▶ Once the flow is linear, start working on playbook(s)
- ▶ Define variables that are needed at the play level as much as possible
- ▶ Work on one task (or bash statement) at a time, converting it to an Ansible task
- ▶ When writing blocks with conditions, make it falsifiable so you can control when the block is run to test with

Convert conditionals

Bash **if**, **for**, and **while** statements are defined at the beginning of the conditional. In ansible they are at the end. Use **block:** to group conditionals together

- ▶ Variables are not quoted in conditionals
- ▶ Conditionals can be joined using **and**, **or** and **()**.
- ▶ Blocks can be nested which each level having its own conditionals
- ▶ Ansible does not have an if-else construct, so use two conditionals, one for each test

Convert conditionals

```
if [ output -gt "1" ]; then
    echo "Greater than one"
else
    echo "Less than one"
```

```
-----
debug:
  msg: "Greater than one"
when: output > "1"
```

```
debug:
  msg: "Less than one"
when: output < "1"
```

Examine System Commands and Arguments

- ▶ Identify Ansible modules that will accomplish each task
 - Note which modules are needed
 - Note which arguments are required for each
- ▶ If a module doesn't exist in Ansible, is one available online (Galaxy, Automation Hub), or possible to create?
 - If yes, include the containing collection in the playbook directory or other location that is available
 - If no, default to shell or command to run the command

Examine System Commands and Arguments

```
if [ add_user == "true" ]; then
/usr/sbin/useradd -d $login_dir -s $login_shell -m -r $user
fi
```

COMMAND

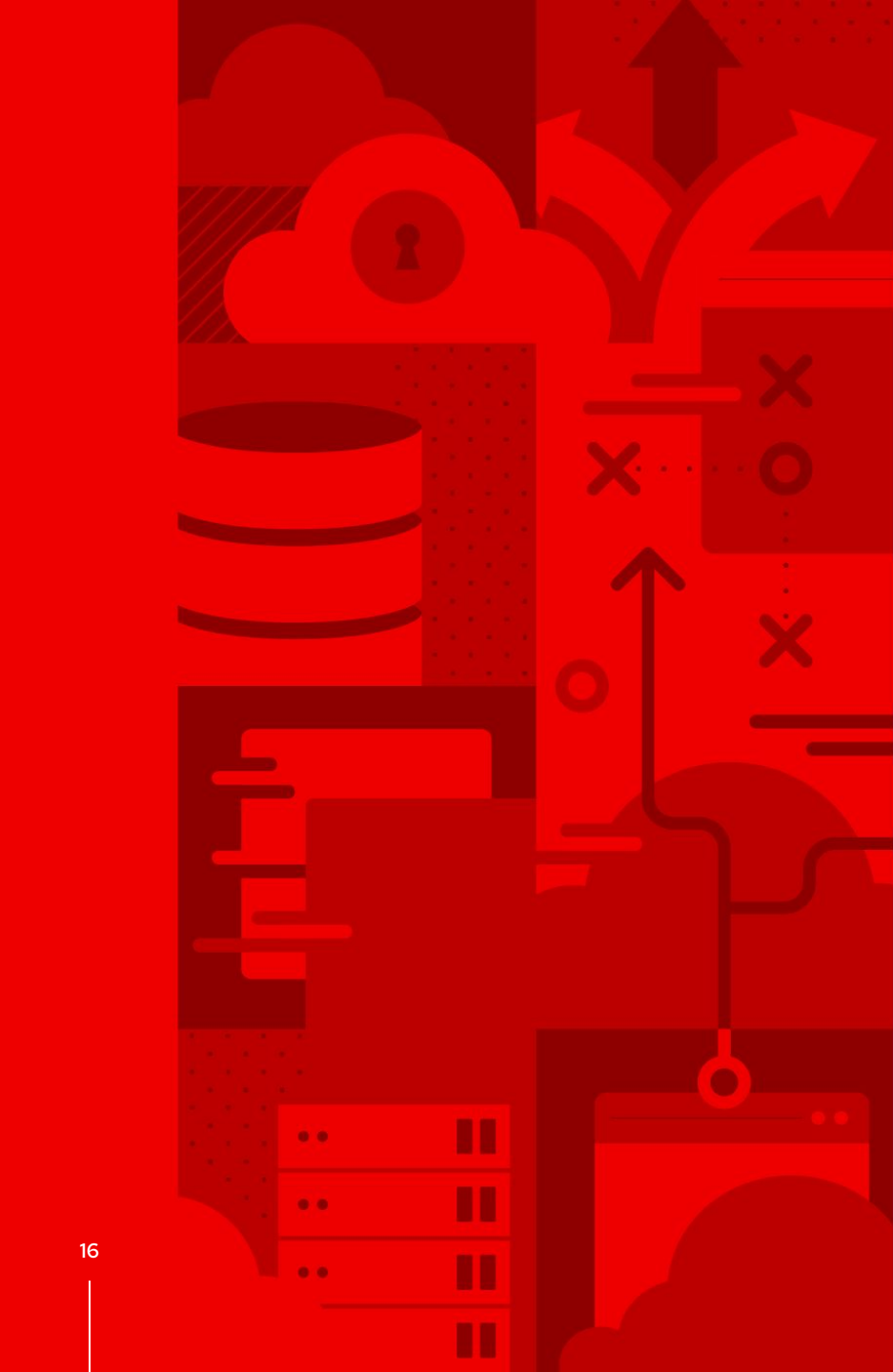
ARGUMENTS

```
-----
- name: Add user
  user:
    home: "{ login_dir }"
    shell: "{ login_shell }"
    create_home: yes
    name: "{ user }"
    system: yes
    state: present
  when: add_user | bool
```

Verify Functionality

Test the Ansible Playbook after every task (or set of tasks) is added.

- ▶ This cuts down on testing at the end of the playbook
- ▶ Assists in verifying system is in proper order for next task




Q&A

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

 [linkedin.com/company/red-hat](https://www.linkedin.com/company/red-hat)

 [facebook.com/redhatinc](https://www.facebook.com/redhatinc)

 [youtube.com/user/RedHatVideos](https://www.youtube.com/user/RedHatVideos)

 twitter.com/RedHat

