ANSIBLE BASICS

INTRODUCTION TO AUTOMATED INFRASTRUCTURE MANAGEMENT WITH ANSIBLE

MANUEL PRINZ & PHILIP R. KENSCHE

INTRODUCTION

THE PROBLEM

- Cloud == Infrastructure
- Cloud users == administrators
- Dynamic infrastructure → challenges

CHALLENGES: REPRODUCIBILITY

- Same setup …
  - … later in the same cloud?
  - … in a different cloud?
  - … in a year from now?
  - … with up-to-date documentation?

CHALLENGES: SCALE-OUT

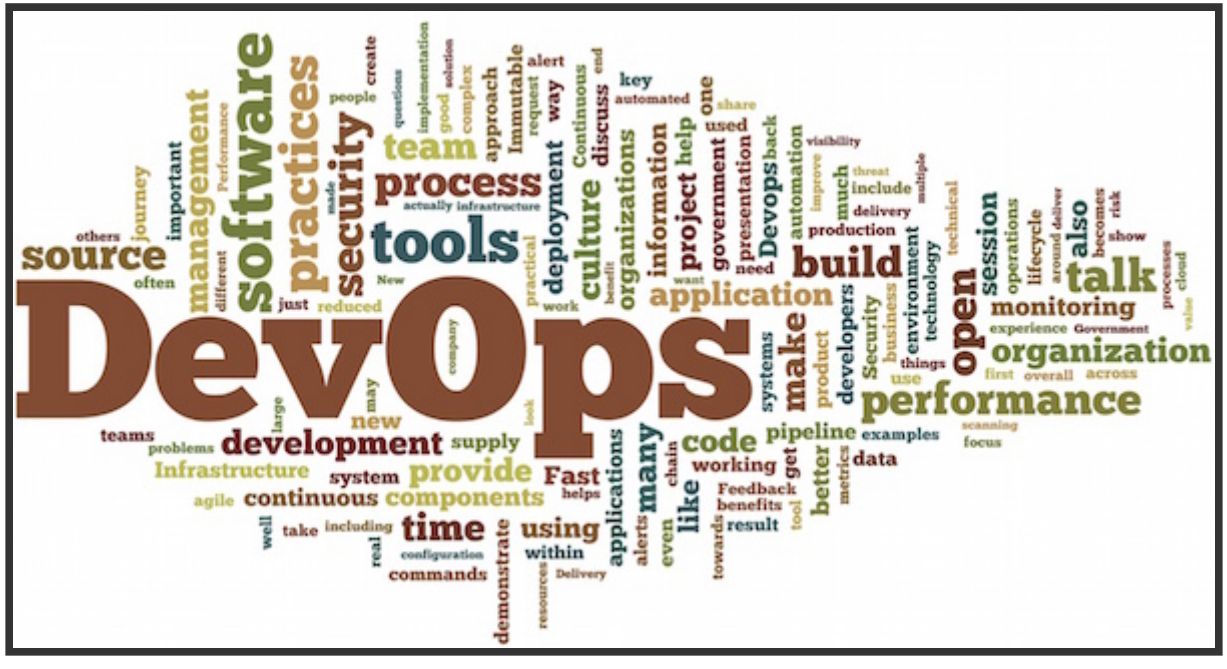
- more machines == more work
- How to set up 200 VMs?

CHALLENGES: QUALITY

- How to ensure that…
  - … the same environment is present everywhere?
  - … changes are deployed to all VMs?
  - … broken setups can be replaced?

HELLO, DEVOPS!

- **Dev**elopment + **Op**eration**s**
- Aim: Remove barriers by taking responsibility
  - Infrastructure becomes part of the application
  - Communication and sharing is key
- Tool-assisted
  - "Infrastructure as Code"
  - "Executable documentation"



Source: DevOpsDays DC 2015

POPULAR DEVOPS TOOLS

Name	Config	Language	OS	Agent?
Chef	DSL (Ruby)	Ruby		Yes
Puppet	Proprietary	Ruby		Yes
SaltStack	YAML	Python		Opt.
Ansible	YAML	Python		No

ANSIBLE: WHY WE CHOSE IT

- Agent-less:
  - SSH and Python are sufficient
- Extensive documentation
- Easy to learn
  - Simple configuration
  - No command-line kung-fu
- Easy to reuse and share
- Version control friendly

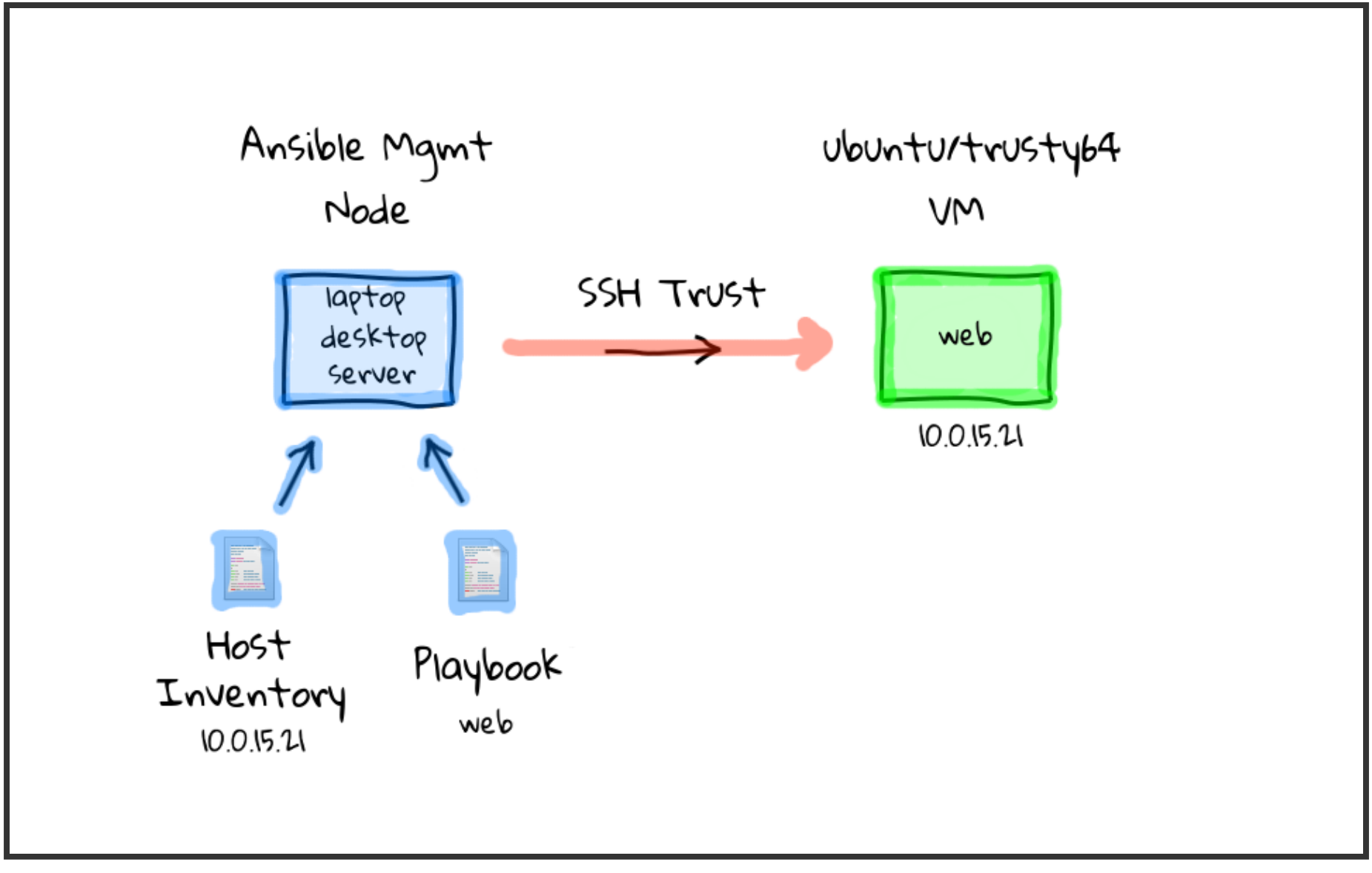
ANSIBLE

CORE CONCEPTS

- "Desired State Engine"
  - Declare the end result
  - Ansible knows the way
- Idempotency (once-only)
  - Repeated execution does not change state
  - Free of side-effects

HOW IT WORKS

1. Connect to remote server (SSH)
2. Transmit code to execute
3. Execute code on remote server
4. Send back status information (JSON)
5. Repeat for next task



Source: Sysadmin Casts, Episode #43

CORE COMPONENTS

- **Playbook**(s)
  - Contains *what* to do
- **Inventory**
  - Contains *where* to do it

INVENTORY (INI)

- Specifies target hosts
- Simplest inventory:
  - Just a list of IPs
- "Virtual" hostname is optional
- Host groups (optional)

Note: YAML is also possible. (Not used in this workshop.)

HOSTS

- List by IPs or host names
  - One per line
- Host-specific variables
  - Follow host name
  - Separated by whitespace
  - Format: `key=value`

A minimal example:

```
172.16.74.164
```

Virtual hostname (optional):

```
vm1 ansible_host=172.16.74.164
vm2 ansible_host=172.16.74.169
vm3 ansible_host=172.16.74.165
```

(More on variables later…)

Dealing with weird naming schemes:

```
node1 ansible_host=dxsc32
node2 ansible_host=gh24nb
```

HOST GROUPS

- Groups are defined as INI "sections"
  - List of hosts below
  - Variables are inherited
- Groups of groups: "`:children`" suffix
- Default groups: `all`

Groups example:

```
frontend ansible_host=72.16.74.164
node1    ansible_host=72.16.74.163
node2    ansible_host=72.16.74.169

[frontends]
frontend

[nodes]
node1
node2
…
[cluster:children]
frontends
nodes
```

DYNAMIC INVENTORY

- Executable program instead of file
- Executed upon each Ansible call

(Not covered in workshop.)

EXCURSION: YAML

YAML

- **Y**et **A**nother **M**arkup **L**anguage
- Simple, and structured
- Readable by humans *and* machines

SIMPLE VALUES

```yaml
---
stringQuoted: "stringValue"
stringValue: someString

integerValue: 1
floatValue: 1.0
floatyStringValue: "1.0"

booleanValue1: true
# ↓ this is Ansible-specific!
booleanValue2: yes
butThisIsAString: "true"
...
```

QUOTING

```yaml
---
DoubleQuoteString: "using 'single quotes' is fine"
SingleQuoteString: 'using "double quotes" is fine'
DoubleDouble: "toil and \"trouble\""

BetterQuoteColons: "or this: will results in an error"
BetterQuoteBraces: "{ NOT interpreted as dict }"
# ↑ same for other YAML chars: [] {} : > |
```

MULTILINE STRINGS

```
multilineString1: |
  Multiline
  keeping newlines

multilineString2: >
  Multiline
  ignoring newlines
```

LISTS (AKA ARRAYS)

```
---
# single-line list
[ package1, package2, package3 ]

---
# Multi-line list
- package1
- package2
- package3
```

ASSOCIATIVE ARRAYS (AKA DICTS, HASHES, ...)

```
---
# Single-line dict
{ key1: value1, key2: value2, key3: value3 }

---
# Multi-line dict
key1: value1
key2: value2
key3: value3
```

NESTED DATA STRUCTURES

```yaml
---
dictOfLists:
  multi:
    - valueA
    - valueB
  inline: [ value1, value2 ]

listOfDicts:
  - keyA: valueA
    keyB: valueB
  - { inline1: value1, inline2: value2 }
```

THERE'S MORE!

- Many other things possible:
  - e.g. type prefixes, anchors, references, …
- Ansible extensions: YAML files …
  - … are processed by a template processor
  - … can `include` other files

MODULES

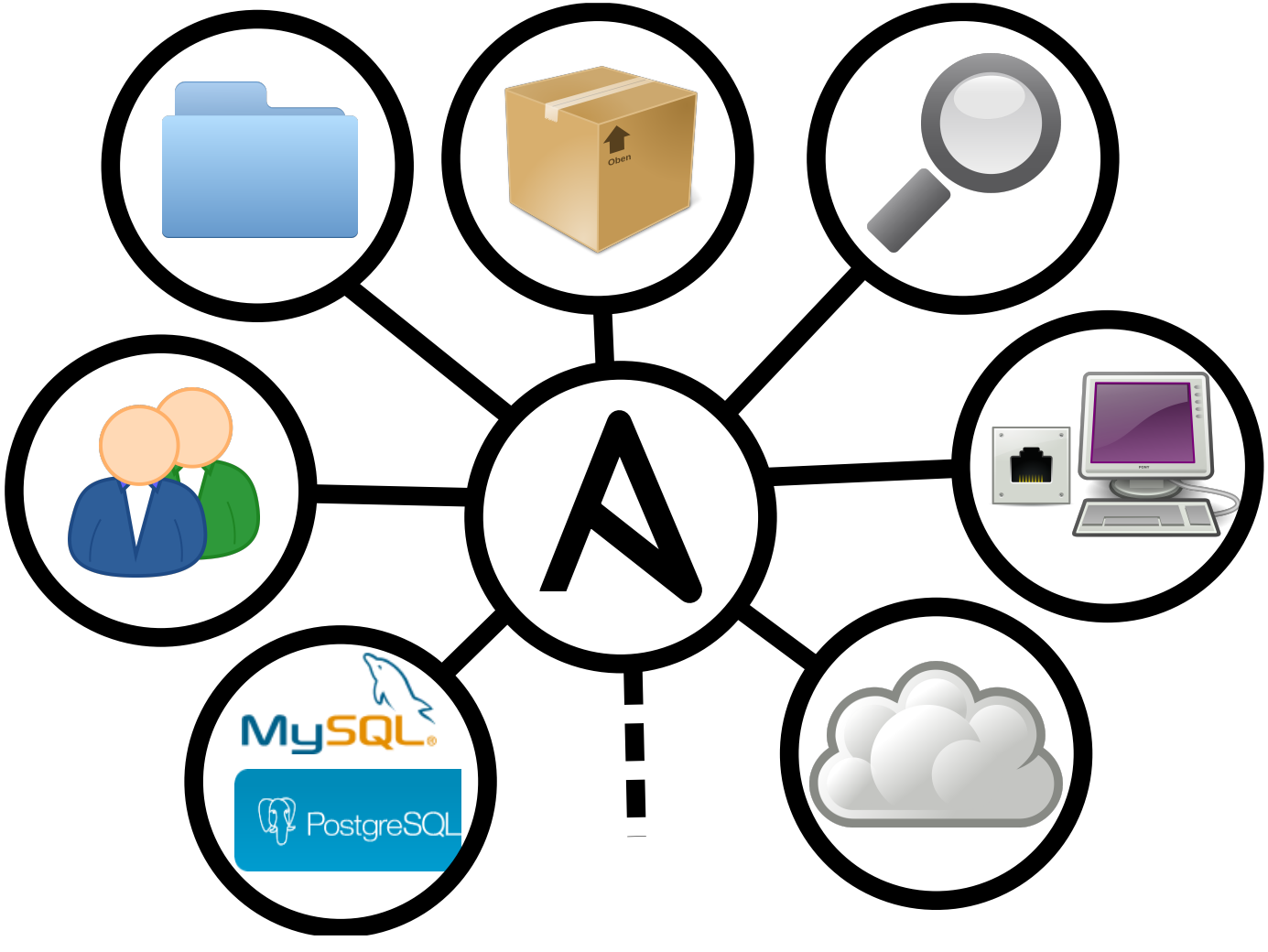
MODULES

- Modules do all the work.
- There are hundreds of them. Literally.
- Even if there are modules to run arbitrary commands…
  - … always prefer already existing modules!
- Idempotent
- Executed on remote host (usually).
- Invoked by play(book)s.

RETURN VALUES

- Modules return data structures containing
  - … status codes: changed, failed, skipped
  - … `stdout` and `stderr`
  - … results
- Can be captured and reacted to

USE CASES



MODULE DOCUMENTATION

- Modules usually need parameters
  - Documentation available via "`ansible-doc`"
    - Up-to-date and matches Ansible version
  - Online documentation
    - Better search capabilities

ANSIBLE HANDS-ON

👉 GET MODULE INFO

Show documentation for the `setup` module:

```
ansible-doc setup
ansible-doc -s setup
ansible-doc --list
```

Questions:

- What did change with the `-s` option?
- What does `-s` stand for?

SETUP

1. Start an Ansible VM on the OpenStack dashboard.
2. Assign a floating IP to the VM.
3. Start VM with the "Debian 9.0 (Stretch)" image.
4. Start VM with the "CentOS 7 (1705)" image.

PREPARE SSH-AGENT [OPTIONAL]

```
eval $(ssh-agent 4h)
ssh-add ~/ssh/key/provided/to/dashboard
```

CREATE A PROJECT DIRECTORY

1. Login to the Ansible VM.
2. Create a new project directory:

```
mkdir -p workshop/day2
```

3. Change into the project directory

```
cd workshop/day2
```

CREATE AN INVENTORY

Create a new file "`inventory`" containing:

```
vm1 ansible_host=$debianHostIp ansible_user=debian
vm2 ansible_host=$centosHostIp ansible_user=centos
```

... with the IPs taken from the dashboard.

CREATE THE CONFIGURATION

Create a file "`ansible.cfg`" containing:

```
[defaults]
; Default inventory file location
inventory = inventory

; Turn off host key checking
host_key_checking = False
```

SUMMARY: WHAT WE LEARNED

- Created the inventory with the host information.
- Set up a basic `ansible.cfg` and set default inventory location.
- Turn off `host_key_checking` to ease working with ephemeral infrastructure.

AD-HOC COMMANDS

👉 PING MODULE

1. Check whether a host responds to ping:

```
ansible -i ./inventory -m ping all
```

```
172.16.74.74 | SUCCESS => {
 "changed": false,
 "ping": "pong"
}
```

2. Did you get the pong?
   - If not, please raise your hand!

👉 SETUP MODULE

1. Collect information about a system:

```
ansible -m setup all
```

```
172.16.73.74 | SUCCESS => {
 "ansible_facts": {
...
}
```

2. Find the keys to the following information:
   - Family of the operating system
   - Your current IPv4 address
3. Quickly browse through the list. Interesting, isn't it?

👉 SHELL / COMMAND MODULES

```
$ ansible -m shell -a "ls -al ~/.bashrc" all
```

```
172.16.74.74 | SUCCESS | rc=0 >>
-rw-r--r--. 1 centos centos 741 17. Mai 11:58 /home/centos/.bashrc
```

- Execute arbitrary commands remotely
  - shell: *with* processing by shell
  - command: *without* shell processing

👉 MODULE PARAMETERS

1. Create a directory via Ansible:

```
ansible -m file -a "name=~/testDir state=directory" all
```

```
172.16.74.74 | SUCCESS => {
 "changed": true,
 …
 "owner": "centos",
 "path": "/home/centos/testDir",
 …
 "state": "directory",
}
```

2. Login to the VM and check the result.
3. What will Ansible do if you rename or delete it? Try!

👉 INSTALL A SOFTWARE PACKAGE

```
ansible -m yum -a "name=cowsay state=present" all
```

- What is the result? Why?

👉 CHANGE USER

1. Use `--become` to `sudo` to the `root` user:

```
ansible -m yum -a "name=cowsay state=present" \
 --become all
```

2. Did the installation work now?
3. Login and check by calling:

```
cowsay "de.NBI Cloud Summer School"
```

👉 CHANGING MODULE BEHAVIOR

1. Try to guess what the following command will do:

```
ansible -m yum -a "name=cowsay state=absent" \
 --become all
```

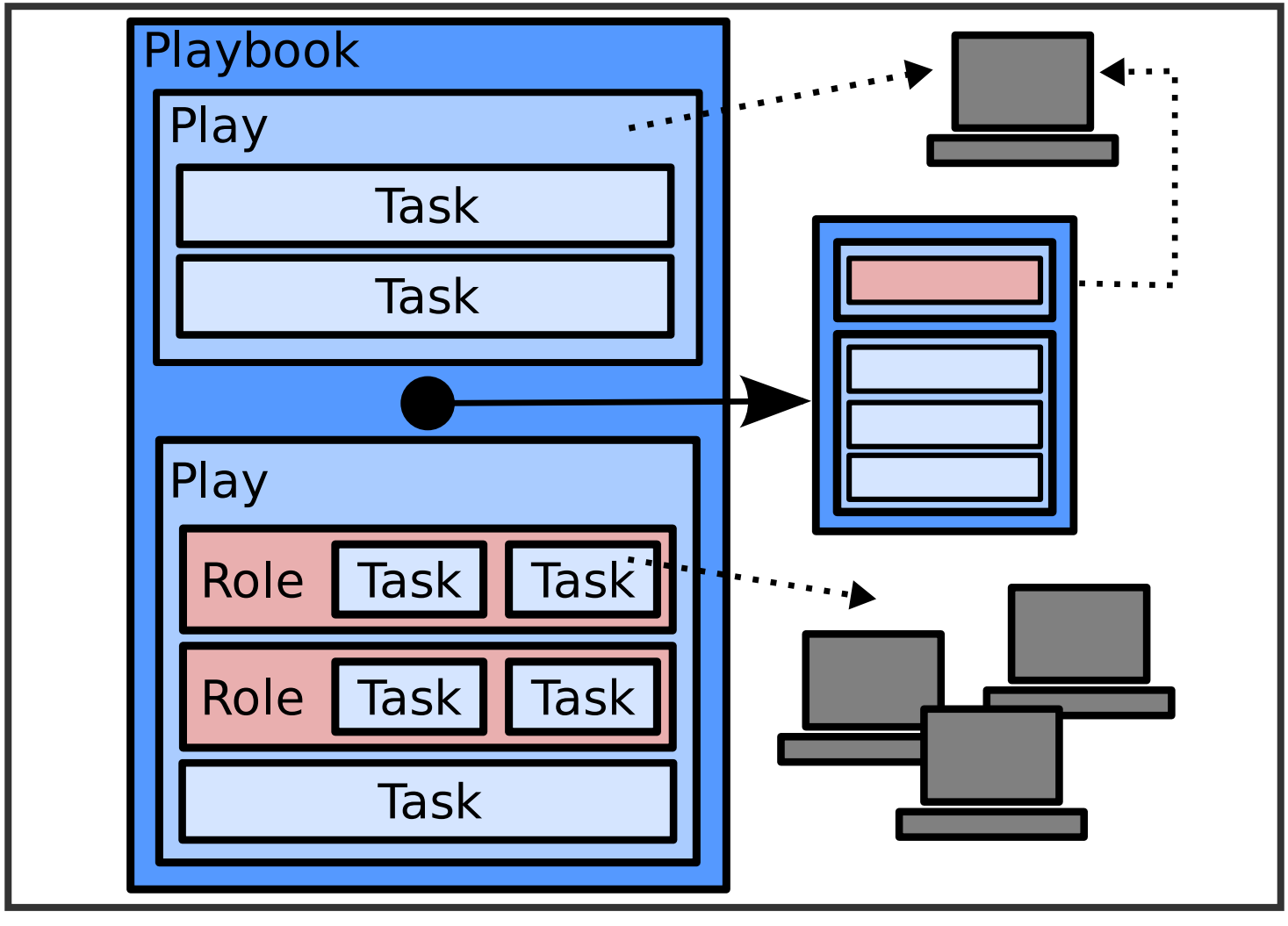
2. Run it!
3. How can you check the result? Do so!

SUMMARY: WHAT WE LEARNED

- Gather facts with the `setup` module.
- Call modules and commands from command-line.
- Configure module behavior via parameters.
- Two Ansible users for SSH and execution.
  - Change to the execution user with `--become`.

PLAYS & PLAYBOOKS

BIG PICTURE


Playbook
Play
Task
Task
Play
Role
Task
Task
Role
Task
Task
Task

TASKS

Smallest unit of action

```
- name: Install screen
  yum:
    name: screen
    state: latest
```

(Cannot stand alone.)

PLAYS

```yaml
---
- name: Common tasks executed on all hosts
  hosts: all
  tasks:
    - name: Install screen
      yum:
        name: screen
        state: latest
```

- Play defines a *list* of tasks for a host group.
- Tasks evaluated in order for each host in a group.

DEPRECATED SYNTAX (BEFORE 2.2)

```
---
- hosts: all
  name: Since 2.2 you should not use varName=varValue syntax
  tasks:
    - name: Install screen
      yum: name=screen state=latest
```

(Some parts of the documentation still use the old syntax.)

RUNNING PLAYS & PLAYBOOKS

```
$ ansible-playbook -i /tmp/inventory site.yml
```

- Convention: "`site.yml`" as "top-most" playbook
  - Call this if there are multiple plays!
- `-i` can be omitted if default is set in `ansible.cfg`

👉 EXERCISE 1: PACKAGE INSTALLATION

- Write a play with one task to install a simple monitoring solution (htop).
- Start with the vm1 (Debian) and use the apt module

SOLUTION I: SITE.YML

```yaml
---
- name: Common tasks executed on all hosts
  hosts: vm1
  tasks:
    - name: Install my favourite monitoring tool
      apt:
        name: htop
        state: latest
```

SOLUTION II: - -BECOME ON THE COMMAND LINE

```
ansible-playbook -i /tmp/inventory --become site.yml
```

SOLUTION III: BECOME : YES IN THE SITE.YML

```yaml
---
- name: Common tasks executed on all hosts
  hosts: vm1
  become: yes
  tasks:
    - name: Install my favourite monitoring tool
      apt:
        name: htop
        state: latest
```

PLAYBOOKS

- Set of plays or playbooks defining your infrastructure
  - plays in one file
  - "include" other plays/playbooks
- Convention: site.yml as playbook entrypoint

```
---
- include: monitoring.yml
- include: ha-webservers.yml
- include: cassandra-cluster.yml
- include: spark-computation-backend.yml
```

👉 EXERCISE 2: THE SAME FOR THE CENTOS VM

- Install "htop" on vm2 (CentOS). Add a second play to the same `site.yml`.
- *Warning*: You won't succeed on the first attempt!

EXERCISE 2: SITE.YML

```
- name: Common tasks executed on CentOS hosts
  hosts: vm2
  become: yes
  tasks:
    - name: Install my favourite monitoring tool
      yum:
        name: htop
        state: latest
```

EXERCISE 2: ADDITIONALLY INSTALL EPEL-RELEASE

```
- name: Common tasks executed on CentOS hosts
  hosts: vm2
  become: yes
  tasks:
    - name: Install EPEL
      yum:
        name: epel-release
        state: latest
    - name: Install my favourite monitoring tool
      yum:
        name: htop
        state: latest
```

SUMMARY: WHAT WE LEARNED

- How to structure and execute Ansible code
  - Tasks, plays & playbooks
  - Single `site.yml`
  - Multiple YAML files + `include`
- Remote package installation using `yum` and `apt`
- Using `ansible_user` and `--become` to log in with sufficient permissions

DEBUGGING

👉 VERBOSE MODE

```
ansible-playbook -i /tmp/inventory site.yml -vvvv
```

- Mode details about ...
  - ... what happens
  - ... return values
  - ... standard output/error of commands

REDUCING EXECUTION TIME AFTER ERROR

TAGS

- `--tags`: Only execute specific tags
- `--skip-tags`: Only skip specific tags

```
- hosts: hostGroup
  tags: playTag
  tasks:
    - name: Some task
      tags: [taskTag1, taskTag2]
    - name: Some other task
      tags: [taskTag3]
```

```
$ ansible-playbook -i /tmp/inventory site.yml \
  --tags "playTag,taskTag1" \
  [--skip-tags "taskTag3"]
```

LIMITS

- Limit to specific host-group
- Upon error a ".retry" file is created

```
$ ansible-playbook -i /tmp/inventory site.yml \
  --limit "hostGroup1,hostGroup2"
```

```
$ ansible-playbook -i /tmp/inventory site.yml \
  --limit @/path/to/site.retry
```

"DEBUG" MODULE

```
- debug:
    msg: Debug the "varOfInterest"
    var: varOfInterest
    verbosity: 0
```

- "verbosity: 3" will print the value only if `-vvv` is set

STRATEGY "DEBUG"

```
- hosts: all
  strategy: debug
```

- Stop into debugger console upon error
- Basic operations
  - r: redo the failed task
  - c: continue with next task
  - q: quit from Ansible

DEBUGGING VARIABLES & TASKS

```
> p vars
> p vars["ansible_hostname"]
> p task      # current task
> p host      # current host
> p result    # current result
```

```
> vars["faulty_variable_value"] = "fixed_value"
> r
```

👉 EXERCISE 3: DEBUGGER

- Debug the playbook exercise-3-debug/site.yml
- You may use "tags" or "--limit" to speed up the debugging.

CHANGE THE SITE.YML

```
- name: Common tasks executed on CentOS hosts
  hosts: vm2
  become: yes
  # Add the debug strategy.
  strategy: debug
  tasks:
    ...
    - name: Install MOTD
      # Tag if you think it is useful
      tags: problem
      copy:
        src: mothd
        name: /etc/motd
```

A DEBUGGER SESSION

```
$ ansible-playbook site.yml --tags problem \
   --limit @/path/to/exercise-3-debug/site.retry
...
TASK [Install MOTD] ****************************************
fatal: [vm2]: FAILED! => {"changed": false, "failed": true, \
   "msg": "src (or content) and dest are required"}
Debugger invoked
(debug) p task.args
{u'name': u'/etc/motd', u'src': u'mothd'}
(debug) task.args["dest"] = "/etc/motd"
(debug) r
fatal: [vm2]: FAILED! => {"changed": false, "failed": true, \
    "msg": "Unable to find 'mothd' in expected paths."}
Debugger invoked
(debug) task.args["src"] = "motd"
(debug) r
```

FIX THE SITE.YML

```
...
- name: Common tasks executed on CentOS hosts
  hosts: vm2
  become: yes
  tasks:
    ...
    - name: Install MOTD
      copy:
        src: motd
        dest: /etc/motd
```

SUMMARY: WHAT WE LEARNED

- Changing verbosity
- Limit execution to
  - specific hosts
  - tagged tasks/plays
- The `debug` *module*
- The `debug` *strategy* and the interactive debugger

VARIABLES

Required to ...

- ... parametrize infrastructure
- ... adapt to variable infrastructure
- ... connect Ansible code from different sources

VARIABLE DEFINITIONS

- Variables can be defined at many places
  - inventory file
  - `./group_vars`
  - `./host_vars`
  - host facts
  - "registered" variables
  - command line

(this is simplified)

DEFINING VARIABLES ON THE CLI

```
# Name value pairs separated by spaces
$ ansible-playbook --extra-vars="name1=value1 name2=value2 ..."
```

```
# JSON, enclosed in {}
$ ansible-playbook --extra-vars='{ "var1": "value1", ... }'
```

```
# YAML or JSON files
$ ansible-playbook --extra-vars="@input.yaml"
$ ansible-playbook --extra-vars="@input.json"
```

USING VARIABLES

```
login_user: "{{ ansible_user }}"
motd: "You are logged in to {{ ansible_hostname }}!"
```

- Ansible requires variables to be quoted with "{{ ... }}"

FILTERS

```
---
example_db_admin: "{{ global_user | default('dbuser') }}"
example_db_admin_pwd: "{{ global_passwd | mandatory }}"
example_db_users: "{{ global_client_users | default([]) | sort }}"
```

- Expressions use the templating engine Jinja2
- Various filters working on
  - lists and dicts
  - IP addresses
  - filenames
  - ...

👉 EXERCISE 4

- Modify play such that the package is provided via a mandatory variable.
- Set variable to "htop" in the `group_vars/all` file.
- Run playbook and check result.
- Run playbook with variable set to "atop" on the command line.

SUBSTITUTE "HTOP" BY VARIABLE REFERENCE

```
...
    - name: Install my favourite monitoring tool
      apt:
        name: "{{ monitoring_tool | mandatory }}"
        state: latest
...
```

GROUP_VARS/ALL

```
---
monitoring_tool: htop
```

SET VARIABLE FROM CLI

```
ansible-playbook -i /tmp/inventory site.yml \
    --extra-vars="monitoring_tool=atop"
```

SUMMARY: WHAT WE LEARNED

- Variables can be defined in many places
- The "{{ }}" Jinja2 notation for accessing variables
- Basic Jinja2 filters
- Variables can be set from the command-line

LOOPS & CONDITIONALS

WHEN

- Run task if condition is true
- when expressions, e.g.
  - comparators: ==, !=, >, <, ...
  - logical operators: and, or, not
  - `is defined`
  - `is undefined`
- Expressions unquoted Jinja2 (no quotes & no {})

```
- name: ...
  when: ansible_pkg_mgr == "yum"
  yum:
    name: htop
    state: latest
```

👉 EXERCISE 5: INSTALL HTOP FOR DIFFERENT OS

- Modify the previous exercise to have a *single* play
  - install "htop" on different OSs
  - install the MOTD
- Check output of "setup" module to find a variable

SITE.YML

```yaml
- name: Common tasks
  hosts: all
  become: yes
  tasks:
    - name: Install my favourite monitoring tool
      when: ansible_pkg_mgr == "apt"
      apt:
        name: "{{ monitoring_tool | mandatory }}"
        state: latest
    - name: Install EPEL
      when: ansible_pkg_mgr == "yum"
      yum:
        ...
```

LOOPING

- `with_items`
- `with_dict`
- Task executed for each entry
- Values successively bound to "item" variable
- Expressions unquoted Jinja2 (no quotes & no {})

```
- with_items: ["a", "b", "c"]
  command: "echo '{{ item }}'"

- with_dict: { a: 1, b: 2, c: 3, d: 4 }
  when: item.value % 2 == 0
  command: "echo '{{ item.key }} is set to {{ item.value }}'"
```

👉 EXERCISE 6: INSTALL MULTIPLE PACKAGES

- Use `with_items` to install multiple packages
- e.g.
  - "htop" for monitoring
  - "links" as text-based browser
  - "curl" and "wget" for downloading data

./GROUP_VARS/ALL

```yaml
---
common_packages:
  - htop
  - links
  - wget
  - curl
```

- `./group_vars/all` are valid for host group "all"
- Low precedence

SITE.YML

```
- name: Common tasks
  hosts: all
  become: yes
  tasks:

    - name: Install packages
      with_items: "{{ common_packages | mandatory }}"
      when: ansible_pkg_mgr == "apt"
      apt:
        name: "{{ item }}"
        state: latest
    ...
```

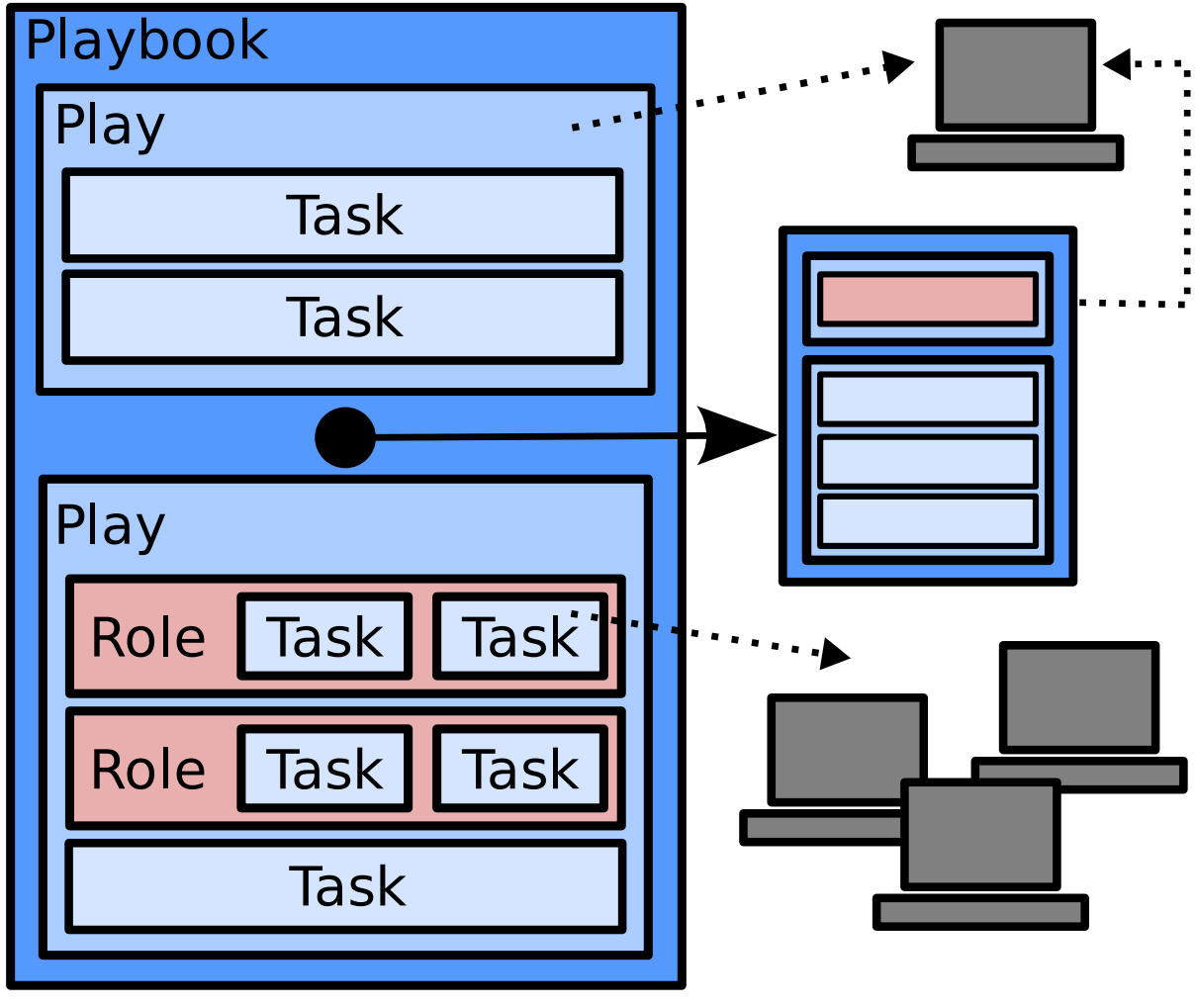
- Same for "ansible_pkg_mgr == 'yum'"
- No loop for EPEL or MOTD tasks

SUMMARY: WHAT WE LEARNED

- Conditionally execute tasks and plays
- Looping over arrays and dictionaries
- Combining `when` & `with_items`/`with_dict`
- Define host-group level data in `./group_vars/`

ROLES

BIG PICTURE (AGAIN)



HOW TO USE ROLES?

- Similar to tasks:
  - Executed on a single host
  - Independent of executions on other hosts
  - One host can fulfil multiple roles
- Similar to plays:
  - Multiple tasks are combined
- Modules relatively independent of other modules
- Well suited for sharing

HOW TO IMPLEMENT ROLES?

```
./roles/$roleName/
             tasks/
             files/
             templates/
             defaults/      # pcdc < inventory
             vars/  # facts < pcdc < registered variable
             ...
```

- Relative to playbook directory
- Each directory contains a `main.yml` as entry point.
- Conveniently created by

```
ansible-galaxy init ./roles/$roleName
```

PLAYS MAP ROLES TO HOST GROUPS

```yaml
- name: Configure the hostGroup
  hosts: hostGroup
  roles:
    - common
    - role: hardening
      configVar: "some value"

  tasks: # executed after "roles"
    - ...
```

👉 EXERCISE 7: EXTRACT ROLE

- Extract your previous MOTD/package installation tasks into a "common" role
- Apply the "common" role to "all" hosts

"SITE.YML"

```yaml
---
- hosts: all
  become: yes
  roles:
    - common
```

"ROLES/COMMON/DEFAULTS/MAIN.YML"

```
## List of packages to install with "apt" or "yum"
common_packages: []
```

- "defaults/main.yml" is just a dictionary of variables.
- not strictly necessary but good practice
  - "defaults/" is the "interface" of the role

"./ROLES/COMMON/TASKS/MAIN.YML"

```yaml
---
- name: Install packages
  with_items: "{{ common_packages | mandatory }}"
  when: ansible_pkg_mgr == "apt"
  apt:
    name: "{{ item }}"
    state: latest

- name: Install EPEL
  when: ansible_pkg_mgr == "yum"
  yum:
  ...
```

Important: Implicit `tasks` block in "tasks/main.yml"

- tasks/main.yml" is just a YAML list of tasks

SUMMARY: WHAT WE LEARNED

- Roles for independently functional configurations
- Plays weave roles together for specific host groups
- Role directory layout
- "defaults/main.yml": A dictionary of variables.
- "tasks/main.yml": A list of tasks.

ADVANCED JINJA2

- A templating engine
  1. Parse Jinja2 expressions from input
  2. Evaluate expressions
  3. Substitute the value for the expression
- In Ansible used in ...
  - ... variable processing
  - ... when & `loop` expressions
  - ... "roles/templates/"

FILTERS

```
new_var: "{{ plain_var }}"
mandatory_var: "{{ some_var | mandatory }}"

is_divisible_by_two: "{{ some_number % 2 == 0 }}"
is_defined: "{{ theVar | defined }}"

list_string: "{{ userlist | sort | join(', ') }}"
```

- Many more built-in filters

JMES PATH

- "XPath for JSON"
- Queries on Python arrays and dictionaries
- Use "json_query()" filter

```
aDict:
  valA:
    - firstA
    - secondA
  valB:
    - firstB
    - secondB
queryResult: "{{ aDict | json_query('*[0]') | sort }}"
# -> ["firstA", "firstB"]
```

JINJA2 CONTROL STRUCTURES

(Mostly used in templates!)

CONDITIONALS

```
{# Only add the "user" variable to the config file, #}
{# if there are users defined. #}
{% if list_of_usernames | length > 0 %}
users = {{ list_of_usernames | join(",") }}
{% endif %}
```

LOOPS

```
{% for user in list_of_users %}

{# user is a dictionary with "name" and "uid" keys #}
{% if user.uid < 500 %}
system_user(user.uid)
{% endif %}

{% endfor %}
```

```
{% for userName, uid in dict_of_users.iteritems() %}
{{ userName }}:{{ uid }}
{% endfor %}
```

👉 EXERCISE 8A: A MOTD TEMPLATE

- Make "files/motd" a template
  - Use the "template" module

EXERCISE 8A: TASKS/MAIN.YML

- Move "files/motd" to "templates/etc/motd.j2"
- Substitute the copy task by a template task in "tasks/main.yml":

```
- name: "Apply MOTD template"
  become: yes
  template:
    src: etc/motd.j2
    dest: /etc/motd
    mode: 0644
```

(Automatically searches in "roles/$roleName/templates")

👉 EXERCISE 8B: ADD EMAIL

- Add mandatory support email ("common_support_email")
- Define the support email address "defaults/main.yml" and leave it empty
- Set the email address in "group_vars/all"

EXERCISE 8B:

- defaults/main.yml

```
# Displayed in MOTD. Mandatory.
common_support_email:
```

- templates/etc/motd.j2

```
For support contact <{{ common_support_email | mandatory }}>.
```

👉 EXERCISE 8C: ADD ANNOUCEMENT

Add one announcement, of the form

```
common_announcement:
      date: 2017-06-27
      content: The de.NBI cloud lifts off!
```

EXERCISE 8C: TEMPLATES/MOTD.J2

```
Recent Announcements:

   {{common_announcement.date}}: {{common_announcement.content}}
```

Or more sophisticated:

```
{% if common_announcement is defined %}

Recent Announcements:

   {{common_announcement.date}}: {{common_announcement.content}}

{% endif %}
```

👉 EXERCISE 8D: MORE ANNOUNCEMENTS

Each announcement should be a dictionary with "date" and "content" keys.

```
common_announcements:
      - date: 2017-06-27
        content: The de.NBI cloud lifts off!
      - ...
```

EXERCISE 8D: ANNOUNCEMENTS.YML

```
common_announcements:
 - date: "yesterday"
   content: "Preparations are going on to get the cloud going."
 - date: "today"
   content: "The de.NBI Cloud is online!"
```

- Can be ...
  - ... placed in `group_vars/all`
  - ... set via CLI

```
$ ansible-playbook -i /tmp/inventory site.yml \
  --extra-vars "@announcements.yml"
```

EXERCISE 8D: DEFAULTS/MAIN.YML

```
# Displayed in MOTD. Each entry a dict with "date" and "content".
common_announcements: []
```

EXERCISE 8D: TEMPLATES/ETC/MOTD.J2

```
{% if common_announcements and common_announcements|length > 0  %}
Recent Announcements:

   {% for announcement in common_announcements %}
  {{ announcement.date }}: {{ announcement.content }}

   {% endfor %}
{% endif %}
```