

# **ANSIBLE PARA DEV+OPS**

**DÍA I – PARTE 2**

# LO QUE VEREMOS HOY

- Mejores prácticas para la escritura de Playbooks
- Técnicas de control de flujo en Playbooks
- Control de versiones (git)
- Probando y depurando nuestros Playbooks
- Labs:
  - Version control
  - Control de flujo con when
  - Variabilidad mediante filtros de Jinja en variables
  - Testeo de nuestros Playbooks

# MEJORES PRÁCTICAS

- Legibilidad del código
- Organización
- Agrupación en roles
- Tagging
- Control de versiones
- Vaults
- Definición de un marco común de trabajo



# LEGIBILIDAD DE CÓDIGO

- El código debe ser lo más simple posible
- En caso de no ser simple o tener casuísticas “extrañas” deben introducirse comentarios
- El indentado debe ser consistente
- Hay que evitar comentarios obvios
- Agrupar bloques que hagan trabajos relacionados
- La nomenclatura que se use para variables/funciones debe ser consistente
- Evitar múltiples niveles de anidado
- Incluir siempre el “state” aunque se use el valor por defecto
- Refactorizar. Refactorizar. Refactorizar.

# ORGANIZACIÓN

- Roles
  - Las carpetas por defecto indican diferentes funciones
  - En defaults se almacenan los valores por defecto de variables
  - En files se almacenan ficheros que pueden ser copiados
  - En templates se almacenan plantillas
  - Meta define metadatos del playbook así como dependencias

```
role/
├── README.md
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   └── main.yml
├── templates
└── tests
    ├── inventory
    └── test.yml
└── vars
    └── main.yml
```

# ORGANIZACIÓN

- Repositorio organizativo:

```
inventories/
  production/
    hosts                               # inventory file for production servers
    group_vars/
      group1                            # here we assign variables to particular groups
      group2
    host_vars/
      hostname1                         # if systems need specific variables, put them here
      hostname2
    #
    #
staging/
  hosts                               # inventory file for staging environment
  group_vars/
    group1                            # here we assign variables to particular groups
    group2
  host_vars/
    stagehost1                         # if systems need specific variables, put them here
    stagehost2
  #
  #
library/
  module_utils/
  filter_plugins/
site.yml
webservers.yml
dbservers.yml

roles/
  common/
  webtier/
  monitoring/
  fooapp/
```

# AGRUPACIÓN EN ROLES

- Si la misma tarea/conjunto de tareas aparece en dos Playbooks diferentes, es susceptible de ser agrupada en un rol
- Importante: variables por defecto
- Importante: documentación del rol
- El comando “ansible-galaxy init”

# TAGGING

- Las tareas / Plays pueden tener un tag
- Estos tags permiten decidir que se ejecuta y que no

```
ansible-playbook example.yml --tags "configuration,packages"
```

```
ansible-playbook example.yml --skip-tags "notification"
```

# TAGGING

- Se puede usar el mismo tag en diferentes tareas
- Se pueden aplicar tags a las Plays pero al final afectan solo a las tareas (pero es útil para no andar escribiendo el tag en todas partes)
- Se pueden aplicar a roles e imports

```
roles:  
  - { role: webserver, port: 5000, tags: [ 'web', 'foo' ] }
```

```
- include_tasks: foo.yml  
  tags: [web,foo]
```

# TAGGING

- Existe un tag especial, “always” , que indica que siempre se ejecute ese paso
- El tag especial “all” hace que esa tarea se ejecute cuando se ejecutan todos los tags
- Los tags especiales tagged y untagged permiten ejecutar o no ejecutar elementos tagueados o sin taguar

# CONTROL DE VERSIONES

- Es importante mantener un control sobre los cambios que hay en el código
- Es importante establecer un protocolo para asegurar la calidad del código que pasa a producción
- Es importante establecer una serie de estándares sobre como se gestiona el código: herramientas, nomenclatura, procesos de test, procesos de aprobación
- Es importante saber que versión de cada rol/playbook se ha aplicado en cada servidor de forma única

# VAULTS

- Los vaults nos permiten almacenar valores “secretos” de forma segura en un repositorio de código
- Es importante que la clave de los vaults solo esté disponible en el host de control y en los equipos en los que se editan/mantienen los vaults
- La clave NUNCA debe estar en un repositorio de código

(asumimos un ciclo automático de despliegue que no pregunta por el password, si no que usa un fichero de clave)

# VAULTS

- Crear un fichero de clave:
  - Usar cualquier herramienta de aleatorización de cadenas o hashear una cadena
- Encriptar un fichero .yml mediante esa clave

```
ansible-vault encrypt test.yml --vault-password-file vault_test
```

- Otros comandos: view, edit, decrypt
- El vault password file se puede indicar en el fichero ansible.cfg para simplificar despliegues

# **DEFINICIÓN DE UN MARCO COMUN DE TRABAJO (DEVELOPMENT STYLE GUIDE)**

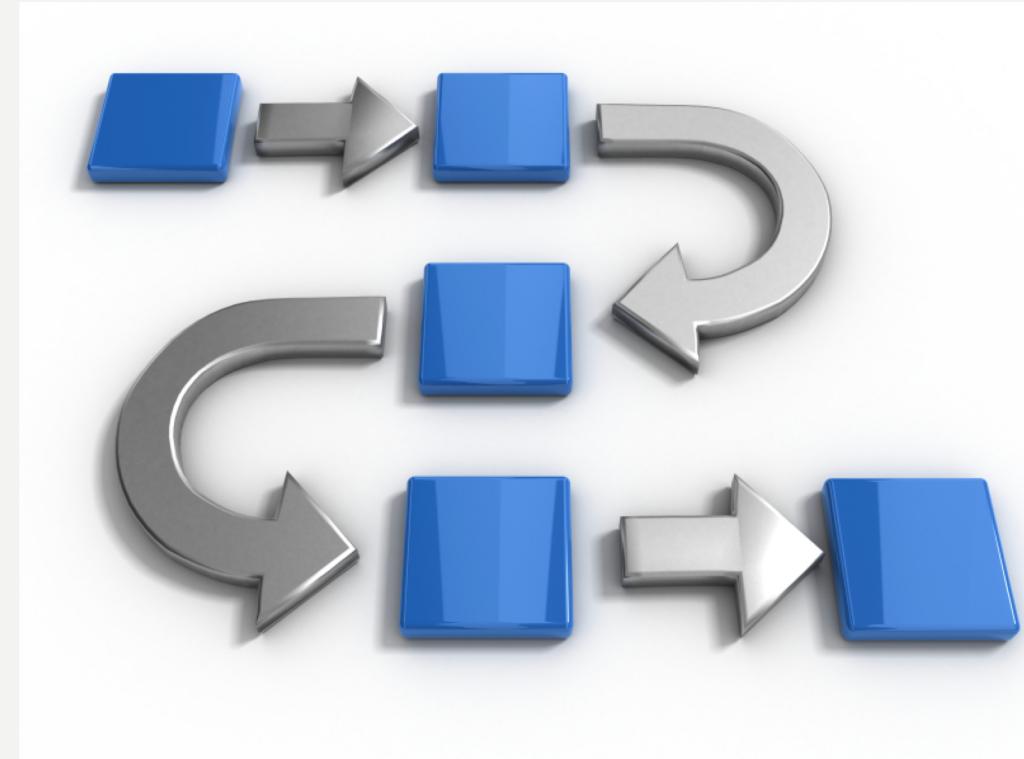
- Facilita el entendimiento entre los miembros de un equipo
- Facilita la incorporación de nuevos miembros al equipo
- A largo plazo, estructura la lectura de código
- Evita errores comunes

# DEVELOPMENT STYLE GUIDE

- ¿TAB o Espacios? ¿cuántos espacios?
- {{variable}} o {{ variable }}
- Separamos-con-guión o separamos\_con\_underscore
- LasVariablesVanAsí o las\_variables\_van\_así
- ESTO\_ES\_UNA\_VAR\_GLOBAL\_O\_LOCAL
- Ejemplo: <https://github.com/whitecloud/ansible-styleguide>
- Guía de cómo hacer una guía:  
[http://docs.ansible.com/ansible/latest/dev\\_guide/style\\_guide/index.html](http://docs.ansible.com/ansible/latest/dev_guide/style_guide/index.html)

# TÉCNICAS DE CONTROL DE FLUJO

- When
- Búcles
- Selección basada en variables
- Registro de variables



# WHEN

- Permite ejecutar una tarea solo cuando una condición se cumple

```
tasks:  
  - name: "shut down Debian flavored systems"  
    command: /sbin/shutdown -t now  
    when: ansible_os_family == "Debian"  
    # note that Ansible facts and vars like ansible_os_family can be used  
    # directly in conditionals without double curly braces
```

# WHEN

- Diferentes formas (para or y and)

```
tasks:  
  - name: "shut down CentOS 6 and Debian 7 systems"  
    command: /sbin/shutdown -t now  
    when: (ansible_distribution == "CentOS" and ansible_distribution_major_version == "6") or  
          (ansible_distribution == "Debian" and ansible_distribution_major_version == "7")
```

```
tasks:  
  - name: "shut down CentOS 6 systems"  
    command: /sbin/shutdown -t now  
    when:  
      - ansible_distribution == "CentOS"  
      - ansible_distribution_major_version == "6"
```

# BÚCLES

- Ansible permite iterar entre:
  - Items
  - Hashes
  - Hashes anidados
  - Ficheros
  - Conjuntos de ficheros
  - Directorios
  - Grupos combinados
  - Elementos / subelementos
  - Secuencias
  - Un aleatorio
  - Hasta que se cumpla una condición
  - Con la primera ocurrencia de un conjunto de reglas
  - El resultado de un comando
  - ...

# BÚCLES

- Ítems (también pueden estar definidos en una variable mediante yaml)

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  with_items:
    - testuser1
    - testuser2
```

# BÚCLES

- Anidados: Ejecuta para cada elemento de la primera lista los elementos de la segunda

```
- name: give users access to multiple databases
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
    with_nested:
      - [ 'alice', 'bob' ]
      - [ 'clientdb', 'employeedb', 'providerdb' ]
```

# BÚCLES

- Hashes

```
---
```

```
users:
  alice:
    name: Alice Appleworth
    telephone: 123-456-7890
  bob:
    name: Bob Bananarama
    telephone: 987-654-3210
```

```
tasks:
  - name: Print phone records
    debug:
      msg: "User {{ item.key }} is {{ item.value.name }} ({{ item.value.telephone }})"
    with_dict: "{{ users }}"
```

# CONTROL DE VERSIONES (GIT)

- Que es Git?
- Concepto de repositorio
  - Local
  - Remoto
- Branching
- Push/pull
- Tags
- Flujos de trabajo con Git



# QUE ES GIT

- Sistema de control de versiones distribuido
- Basado en grafos
- Permite ramas, tags...

# REPOSITORIO

- Conjunto de ficheros versionados
- Local:
  - Cualquier directorio donde se ha ejecutado “git init .”
- Remoto:
  - Servidor que almacena una copia de un repositorio y centraliza las aportaciones de todos los desarrolladores

# BRANCHES / BRANCHING

- Las ramas nos permiten mantener diferentes momentos del desarrollo de forma independiente
- Se pueden mantener ramas en local que no existen en el repositorio remoto
- Se pueden unir (merge) ramas para aplicar los cambios de una en la otra

# PUSH / PULL

- PUSH: Enviar el contenido local al servidor remoto
- PULL: Recibir los cambios que hay en el servidor remoto y unirlos a los cambios locales

# TAGGING

- Se pueden añadir tags para indicar puntos en el ciclo de vida del código (versiones)

```
$ git tag -a v1.4 -m "my version 1.4"  
$ git tag  
v0.1  
v1.3  
v1.4
```

# FLUJOS DE TRABAJO EN GIT

- Equivalen a las development guidelines
- Cada organización debe asumir el suyo
- Populares:
  - Gitflow: <https://datasift.github.io/gitflow/IntroducingGitFlow.html>
  - Github flow: <https://guides.github.com/introduction/flow/>

# PROBANDO Y DEPURANDO PLAYBOOKS

- Como funciona la ejecución de un playbook
- Mensajes de debug
- Playbook debugger



# EJECUCIÓN DE UN PLAYBOOK

- I. Análisis de código
2. Convertir conjunto de tareas de una play en un script de python
3. Enviar el script a los nodos remotos y ejecutarlos
4. Analizar resultado y seguir/fallar

# MENSAJES DE DEBUG

- Permiten sacar información durante la ejecución del playbook
- Son útiles para entender las salidas de algunos módulos y para ver que está pasando
- No son útiles para encontrar errores

```
# Example that prints the loopback address and gateway for each host
- debug:
  msg: "System {{ inventory_hostname }} has uuid {{ ansible_product_uuid }}"

- debug:
  msg: "System {{ inventory_hostname }} has gateway {{ ansible_default_ipv4.gateway }}"
when: ansible_default_ipv4.gateway is defined
```

# PLAYBOOK DEBUGGER

- Nos permite depurar paso a paso un playbook
- Se activa indicando “strategy: debug” en una play
- Cuando un paso falla indica con un prompt varias acciones
  - Podemos usar p para sacar el valor de una variable / tarea
    - p task
    - p vars
    - p task.args
    - p host
    - p result

**LABS**

---

Darse de alta en github

---

Iniciar el primer repositorio

---

Compartir el repositorio con el equipo (dependiendo del tamaño de la clase se harán varios grupos)

# LABS

- Elaborar un marco de trabajo del equipo
- El marco de trabajo debe:
  - Especificar como se escribe el código
  - Establecer un flujo de gestión y aprobación de versiones en git
  - Establecer un flujo de pruebas mínimo que se considere aceptable

# LABS

- Inicializar máquinas de labs “dia2-lab I”
- Crear el inventario estático de las 3 máquinas que se creen de forma que una sea el grupo database y las otras dos el grupo www
- Crear un playbook que instale nginx en las máquinas www y despliegue una página de bienvenida, y que instale mysql en las máquinas bbdd
- Segregar el playbook mediante roles
  - Rol nginx
  - Rol mysql
- Preparar el playbook para que permita el uso de tags en base de la función
- Aplicar las normas del marco de trabajo a éste playbook
- Destruir máquinas de labs “dia2-lab I”

# LABS

- Crear un repositorio de organización con la estructura definida en el marco de trabajo
- Compartir el repositorio de organización con los compañeros de equipo
- Integrar los roles y playbooks que hemos implementado en el lab anterior en el repositorio de organización
- Cada miembro del equipo creará un entorno mediante el entorno “dia2-lab2” y definirá un fichero de inventario que segregue dos máquinas www y una de bbdd
- Se probarán los playbooks para los diferentes entornos /tags
- Destruir los entornos “dia2-lab2”

# LABS

- Lanzar el entorno dia2-lab3
- Crear el inventario para este entorno
- Crear un rol que instale un set de paquetes “mínimo” para poder trabajar en el sistema usando bucles
- Aplicar el rol mediante un playbook
- Destruir el entorno dia2-lab3