

# Ansible

- [Список используемых источников](#)
- [Molecule: Тестирование ролей](#)
- [Использование переменных внутри других переменных](#)
- [Расстановка отступов](#)
- [Использование нескольких строк в переменных](#)
- [Использование фильтров для управления данными](#)
- [Ansible galaxy](#)
  - [Управление зависимостями](#)
- [Инвентарь](#)
  - [Динамический инвентарь](#)
- [Пишем роли ansible не ломая прод — особенности check\\_mode или как правильно его готовить](#)
- [Переменные среды](#)
- [Callback plugins](#)
- [Отладка](#)
  - [Задачи по отладке](#)
- [Jinja](#)
  - [Макросы](#)



# Список используемых источников

- [Good practices for ansible](#)
- [Ansible naming conventions](#)
- [Looking up secrets from Hashicorp Vault](#)
- [Galaxy Installation with Ansible](#)
- [How to use Ansible Galaxy](#)
- [Ansible role argument specification](#)
- [Пишем роли ansible не ломая прод — особенности check\\_mode или как правильно его готовить](#)

# Molecule:

# Тестирование ролей

## Список используемых ИСТОЧНИКОВ:

- [Ansible Molecule project](#)
- [Разработка и тестирование Ansible-ролей с использованием Molecule и Podman](#)
- [Тестирование Ansible с использованием Molecule с Ansible в качестве верификатора](#)
- [Ansible Testing Using Molecule with Ansible as Verifier](#)
- [Тестирование ролей Ansible с помощью Molecule в Ubuntu 18.04](#)
- [Инструкция: как тестировать ansible-роли и узнавать о проблемах до продакшена](#)
- [Testing your Ansible roles with Molecule](#)
- [Testing Ansible roles and playbooks with Molecule](#)
- [Ansible Collections: Role Tests with Molecule](#)

# Использование переменных внутри других переменных

## Дополнение "vars"

```
set_fact:
```

```
variable: '{{ vars['my_' + variablename + '_variable'] }}'
```

# Расстановка отступов

## Yaml

```
app:
  config1:
    base:
      {{ service1.company.backend | to_nice_yaml(indent=2) | trim | indent(6) }}
  config2:
    node:
      {{ service1.company.addr | to_nice_yaml(indent=2) | trim | indent(6) }}
```

# Использование нескольких строк в переменных

## Объединение строк методом **join**

Этот способ позволяет использовать стандартный метод строки - **join**.

```
ansible -m debug -a msg="{{ '-'|join((var1, var2, var3)) }}" localhost
```

## Список используемых источников

- [Understanding multi line strings in YAML and Ansible \(Part I - YAML\)](#)

# Использование фильтров для управления данными

## Определение типа данных

Появилось в версии 2.3

Если вы не уверена какой тип используется в переменной, вы можете использовать фильтр `type_debug` для его определения. Это может быть необходимо в случае когда вам требуется особый тип переменной.

```
{{ myvar | type_debug }}
```

## Список используемых источников

- [Using filters to manipulate data](#)
- [How to use Ansible 'selectattr' Filter](#)
- [Two ways to correctly deep merge Ansible and nested dict variables](#)
- [Jinja2 filters](#)



- [Manipulating data](#)

# Ansible galaxy

# Управление зависимостями

## Установка коллекций

```
ansible-galaxy collection install <FCN>
```

## Установка в отдельный каталог

```
ansible-galaxy collection install --collections-path collections <FCN>
```

**<FCN>** - Имя коллекции вместе с её полным путём (fully-qualified collection name)

# Инвентарь

# Динамический инвентарь

## Список используемых источников

1. [Динамические инвентории Ansible](#)
2. [Working with dynamic inventory](#)

# Пишем роли ansible не ломавая прод — особенности check\_mode или как правильно его ГОТОВИТЬ

В данном опусе я попытаюсь рассказать как можно писать роли и плэйбуки такими, чтобы они отыгрывали без падений при запуске с флагом `--check`. Зачем вот это всё: вы в команде адептов инфраструктуры как код и не только раскатываете свою инфраструктуру, но и обновляете ее и хотите быстро восстанавливать, а это значит что вы должны обеспечивать одну из ключевых концепций ansible - идемпотентность повторных запусков. Для этого вы вынуждены не только писать роли, применяя их в лабораторных условиях, но и применения их на реальной инфраструктуре, о том как дебажить при этом ямлы написано например [здесь](#). Применять код который сделает множество изменений в продакшн - такое себе удовольствие не для слабонервных, поэтому и мы будем пускать его предварительно с флагом `--check` - в холостую, желая посмотреть те изменения которые будут внесены в целевую инфраструктуру. То есть, если вы собираетесь реконфигурировать сервис в одном конфиге и перезапустить-перегрузить его, то вы должны увидеть только 2 изменения (changed) по итогам запуска.

Итак, как этого добиться.

# Игнорируем игнорирование

Конечно же можно заигнорить таски с ошибками выставив `ignore_errors: true`, но это путь к непредсказуемому поведению на целевом хосте и пропуску не только единичных задач, но и блоков и плэйбуков, в рамках холостого запуска. К чему это приведет - это приведет к потенциальной необработанной ошибке. Как это исправить - определить логическую переменную и использовать ее в качестве флага игнорирования:

```
- set_fact:
    your_vars_ignore: true
- debug:
    var: your_vars_ignore
    ignore_errors: "{{ your_vars_ignore }}"
```

Использовать в качестве таковых переменных встроенные переменные `ansible`, например `ansible_check_mode`, то есть при запуске с флагом `--check` игнорировать ошибки - поможет вам в тех случаях когда нецелесообразно вычислять промежуточную переменную и выполнение игнорируемого кода предсказуемо.

```
- debug:
    msg: "no errors in check mode"
    ignore_errors: "{{ ansible_check_mode }}"
```

# Никаких холостых прогонов

С другой стороны даже такое игнорирование часто не обеспечивает полноту выполнения холостого запуска кода. В тех случаях когда блок кода не вносит изменения на целевую систему, например определение переменных, скачивание на контроллер архивов для их распаковки, стоит игнорировать `check_mode`:

```
---  
- set_fact:  
    always_var: "this var is defined always"  
    check_mode: false
```

## Хороший shell - он как модуль

Мы все частенько пренебрегаем использованием модулей `command` и `shell` не исках найдя нужного модуля `ansible` или используя специфичные утилиты командной строки. Так вот использование этих модулей с флагом `check` будет практически всегда выдавать изменение. Этим можно и нужно гибко [управлять](#).

## Учимся правильно падать

`Ansible` позволяет определить, что означает “сбой” в каждой задаче, используя `failed_when` условие. Как и во всех условных обозначениях в `Ansible`, списки из нескольких `failed_when` условий объединены неявным `and`, что



означает сбой задачи только при выполнении всех условий. Если вы хотите вызвать сбой при выполнении любого из условий, вы должны определить условия явным `or` .

Вы можете проверить наличие сбоя, выполнив поиск слова или фразы в выходных данных команды или на основе кода возврата:

```
- name: Fail task when the command error output prints FAILED
ansible.builtin.command: /usr/bin/example-command -x -y -z
register: command_result
failed_when: "'FAILED' in command_result.stderr"
```

```
- name: Fail task when both files are identical
ansible.builtin.raw: diff foo/file1 bar/file2
register: diff_cmd
failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

## Учимся правильно изменяться

Ansible позволяет определить, когда конкретная задача “изменила” удаленный узел, используя `changed_when` условие. Это позволяет вам определить, на основе кодов возврата или выходных данных, следует ли сообщать об изменении в статистике Ansible и следует ли запускать обработчик или нет. Как и во всех условных обозначениях в Ansible, списки из нескольких `changed_when` условий объединены неявным `and` .

```
---
- name: Report 'changed' when the return code is not equal to 2
  ansible.builtin.shell: /usr/bin/billybass --mode="take me to the river"
  register: bass_result
  changed_when: "bass_result.rc != 2"

- name: This will never report 'changed' status
  ansible.builtin.shell: wall 'beep'
  changed_when: False
```

```
- name: Combine multiple conditions to override 'changed' result
  ansible.builtin.command: /bin/fake_command
  register: result
  ignore_errors: True
  changed_when:
    - '"ERROR" in result.stderr'
    - result.rc == 2
```

# Выводы

Использование описанных выше способов позволяет выполнить следующие требования к скриптам конфигурирования:

- при повторном запуске скрипта состояние целевой системы не будет изменено
- при запуске скрипта с флагом `--check` выполняются все задачи
- запуск скрипта с флагом `check` не приводит к изменению на целевой инфраструктуре

# Переменные среды

- `ANSIBLE_MAX_DIFF_SIZE=104857600` - ?????????? ?????? ??????????  
???????? ?????? ?????????

# Callback plugins

## Список используемых ИСТОЧНИКОВ

- [Creating an Alerting Callback Plugin in Ansible - Part I\( ansible\\_stdout\\_compact\\_logger\)](#)

# Отладка

Отладка

# Задачи по отладке

## Список используемых ИСТОЧНИКОВ

- [Debugging tasks](#)

Jinja

# Макросы

## Использование макросов других из других файлов

```
{% from "<macro_file>" import <macro_name> with context %}
```

`with context` может работать не на всех версиях jinja2.

### Macro\_file:

```
{% macro macro_name() %}  
...  
{% endmacro %}
```

## Список используемых источников

- [Include and import](#)