Ansible для управления конфигурацией Windows. История успеха

* [Блог компании Veeam Software](https://habr.com/ru/company/veeam/),
* [Системное администрирование](https://habr.com/ru/hub/sys_admin/),
* [Программирование](https://habr.com/ru/hub/programming/),
* [.NET](https://habr.com/ru/hub/net/),
* [IT-инфраструктура](https://habr.com/ru/hub/it-infrastructure/)

На одной из встреч питерского сообщества .Net разработчиков SpbDotNet Community мы пошли на эксперимент и решили рассказать о том, как можно применять подходы, давно ставшие стандартом в мире Linux, для автоматизации Windows-инфраструктур. Но дабы не доводить всё до голословного размахивания флагом Ansible, было принято решение показать это на примере развёртывания ASP.Net приложения.

Быть спикером вызвался Алексей Чернов, Senior Developer команды, разрабатывающей библиотеки UI-компонентов для наших проектов. И да, вам не показалось: выступать перед .Net аудиторией пошёл JavaScript разработчик.

Кто заинтересовался итогом такого эксперимента, милости просим под кат за расшифровкой.

Привет) Тут уже немного заспойлерили и сказали, что я фронтендер, так что можете уже расходиться =) Меня зовут Алексей, я занимаюсь всяким-разным про веб разработку довольно давно. Начинал с Perl, потом был PHP, немного RoR, немного того, чуть-чуть этого. А потом в мою жизнь ворвался JavaScript, и с тех пор я занимаюсь практически только этим.

Помимо JS, последнее время я пишу довольно много автотестов (причём на том же JS), и поэтому приходится заниматься автоматизацией деплоя тестовых стендов и самой инфраструктуры для них.

**Предыстория**

Два года назад я оказался в Veeam, где разрабатывают продукты под Windows. В тот момент я очень удивился, но оказалось, что и так бывает =). Но больше всего меня удивил непривычно низкий уровень автоматизации всего, что связано с деплоем, с разворачиванием приложений, с тестированием и т.д.

Мы — те, кто разрабатывает под Linux — уже давно привыкли, что всё должно быть в Docker, есть Kubernetes, и всё разворачивается по одному клику. И когда я оказался в среде, где всего этого нет, это шокировало. А когда я начал заниматься автотестами, я понял, что это всего лишь 20% успеха, а всё остальное это подготовка инфраструктуры для них.

  
*Мои ощущения в начале*

**Текущие условия**

Немного расскажу про то, как у нас всё устроено, что нам приходится автоматизировать и чем мы занимаемся.

У нас есть куча разных продуктов, большинство из них под Windows, есть несколько под Linux и даже что-то под Solaris имеется. Ежедневно собирается довольно много билдов для всех продуктов. Соответственно, надо это всё раскатывать в тестовых лабах, как для QA, так и для самих разработчиков, чтобы они могли проверять интеграцию приложений. Всё это требует огромной инфраструктуры из множества железных серверов и виртуальных машин. А ещё иногда мы проводим тестирование производительности, когда надо поднимать сразу по тысяче виртуалок и смотреть, насколько быстро наши приложения будут работать.

**Проблемы**

Конечно же, на первых этапах (читай, давно) при попытке автоматизировать всё в лоб был использован PowerShell. Инструмент мощный, но скрипты деплоя получаются крайне переусложнёнными. Другая проблема заключалась в отсутствии централизованного управления этим процессом. Какие-то скрипты запускались локально у разработчиков, какие-то на виртуалках, созданных ещё в эпоху мамонтов, и т.д. Как итог: трудно был получить единый результат и понять, что работает, а что нет. Ты приходишь на работу, открываешь браузер — сервер недоступен. Почему недоступен, что случилось, где сломалось — было совершенно не понятно. Не было единой точки входа, и приходилось искать правду по рабочим чатикам, и хорошо, если кто-то отвечал.

Другая проблема, не такая очевидная — это новички. Им было сложно. Первую неделю работы они просто вникали в происходящее. Мы по привычке жили с этим, успокаивая себя тем, что жизнь — штука сложная и надо смириться. Понять и простить, так сказать.

Но в какой-то момент нашли внутренние силы перебороть это и посмотреть по сторонам. Вероятно, как-то с этим можно справиться.

  
*Первый шаг на пути решения проблемы — её принятие*

**Подбор решения**

Когда не знаешь, что делать, посмотри, что делают другие.

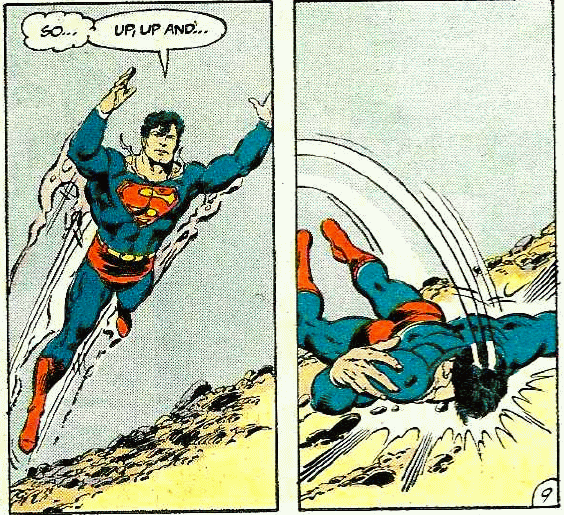
И для начала мы составили свой список требований к тому, что мы хотим получить в конце.

* Единая кодовая база. Все скрипты деплоя должны лежать в одном месте. Хочешь что-то развернуть или посмотреть, как оно разворачивается: вот тебе репозиторий, иди туда.
* Все знают, как это работает. Должны исчезнуть вопросы а-ля "Я не понимаю, как это развернуть, поэтому второй день не могу закрыть багу".
* Возможность запуска по кнопке. Нам нужно иметь возможность контроля деплоев. Например, какой-то веб интерфейс, куда заходишь, нажимаешь кнопку, и разворачивается нужный продукт на нужном хосте.

Убедившись, что данный список покрывает минимум необходимых и достаточных требований для нашего счастья, мы начали пробовать. Традиционно, первым делом попытались решать проблемы методом лобовой атаки. У нас много PowerShell скриптов? Так давайте объединим их в один репозиторий. Но проблема не в том, что скриптов было слишком много, а в том, что разные команды делали одно и тоже разными скриптами. Я походил по разным командам, послушал их требования, собрал одинаковые скрипты, попытался их как-то более-менее причесать и параметризировать, а потом сложил в единый репозиторий.

**Fail:** Попытка провалилась. Во-первых, мы стали очень много спорить, почему мы делаем так, а не этак. Почему был использован это метод, а не какой-то другой и т.д. И как следствие, появилось много желающих переделать всё "как надо", по принципу "Я сейчас форкнусь и всё за вас перепишу". А объединить ветки с таким подходом, конечно, не удастся.

Попытка номер два: предполагалось взять наш CI-сервер (TeamCity), сделать на нём некие шаблоны и с помощью наследования закрыть основную проблему из первой попытки. Но, как вы могли сразу догадаться, тут нас тоже ждал **Fail:** Можно использовать шаблон только последней версии, а значит, мы не добьёмся необходимой версионности. И следствие большого количества команд — шаблонов становилось очень много, управлять ими становилось всё сложнее, а на горизонте отчётливо виднелось новое болото.



Устав падать лицом о землю при всякой попытке взлететь, было решено ещё раз сесть и крепко подумать. Итак, у нас есть большая куча ps-скриптов с одной стороны и огромное количество виртуалок с другой. Но мы ошибались, т.к. корень проблемы был совершенно не в этом. Проблема была в том, что между этими вещами всегда был человек. Неважно, будь то разработчик, тестировщик или ещё кто-то, в голове всегда происходила примерно такая логическая цепочка:

* Так, мне нужна виртуалка для тестов
* Ага, вот у нас есть пул хостов
* А вот нужный мне скрипт, сейчас запущу его, и всё случится

С осознанием такой вроде бы простой вещи общая проблема заиграла новыми красками. Оказалось, что все наши боли от отсутствия единого описания нашей инфраструктуры. Она была в головах людей, её создававших, они сидели в разных отделах, не стремились как-то её документировать, и вообще каждый жил в своём отдельном государстве.  
В этот момент мы пришли к тому, что ответ на все наши проблемы это:

**Infrastructure as a Code**

Это именно о том, что вся наша инфраструктура должна быть описана в коде и лежать в репозитории. Все виртуалки, все их параметры, всё, что туда устанавливается — всё надо описать в коде.

Возникает законный вопрос — зачем?

Отвечаем: такой подход даст нам возможность применять лучшие практики из мира разработки, к которым мы все так привыкли:

* Version control. Мы всегда сможем понимать, что и когда поменялось. Больше никаких пришедших из ниоткуда или пропавших в никуда хостов. Всегда будет понятно, кто внёс изменения.
* Code Review. Мы сможем контролировать процессы деплоя, чтобы одни не ущемляли других.
* Continuous Integration.

**Выбор инструмента**

Как все мы знаем, инструментов для управления конфигурациями довольно много. Мы свой выбор остановили на Ansible, поскольку он содержит набор фич, необходимых именно нам.  
Прежде всего, от системы автоматизации мы хотим, не чтобы запускались какие-то инсталляторы, что-то куда-то мигрировало и т.д. Прежде всего от такой системы мы хотим, чтобы после нажатия одной кнопки мы видели UI необходимого нам приложения.

Поэтому ключевая для нас фишка — идемпотентность. Ansible не важно, что было с системой "до". После запуска нужного плейбука мы всегда получаем один и тот же результат. Это очень важно, когда ты говоришь не "Установи IIS", а "Тут должен быть IIS", и тебе не надо думать, был он там до этого, или нет. Скриптами такого достичь очень сложно, а плейбуки Ansible дают такую возможность.

Также надо упомянуть безагентность Ansible. Большинство систем автоматизации работает через агентов. В этом есть много своих плюсов — например, лучший перфоманс — но нам было важно отсутствие агента, чтобы систему не приходилось дополнительно как-то подготавливать.

PowerShell:

$url = "http://buildserver/build.msi"

$output = "$PSSscriptRoot\build.msi"

Invoke-WebRequest -Uri $url -OutFile $output

Ansible:

name: Download build

hosts: all

tasks:

name: Download installer

win\_get\_url:

url: "http://buildserver/build.msi"

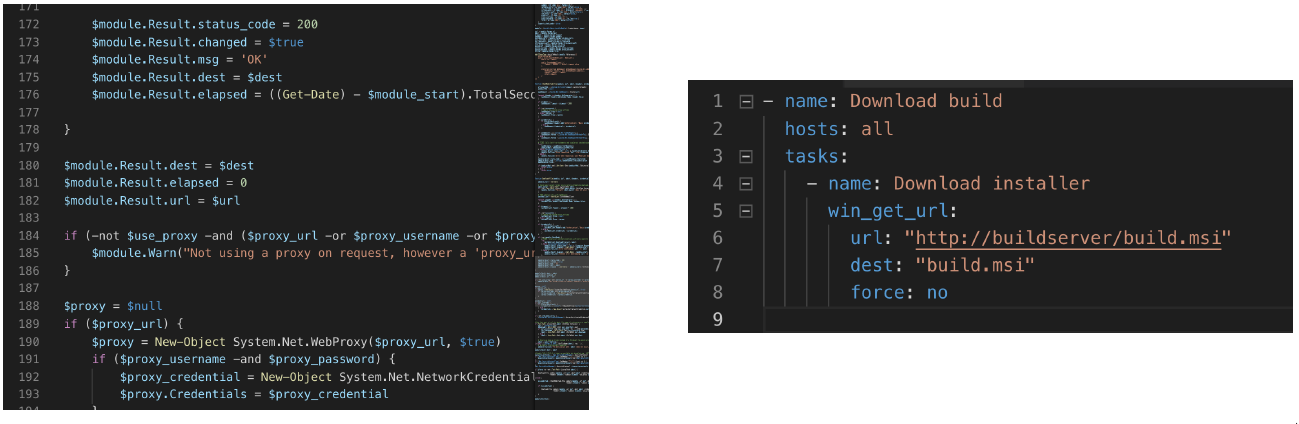
dest: "build.msi"

force: no

Тут мы видим, что в базовом примере ps-скрипт будет даже лаконичнее, чем Ansible плейбук. 3 строчки скрипта против 7 строчек плейбука ради того, чтобы скачать файл.

Но, Петька, есть нюанс (с). Как только мы захотим соблюсти принцип идемпотентности и, например, быть уверенным, что файл на сервере не изменился и его не надо качать, в скрипте придётся реализовать HEAD-запрос, что добавляет примерно 200 строчек. А в плейбук — одну. Модуль Ansible win\_get\_url, который делает за вас все проверки, содержит 257 строк кода, которые не придётся вставлять в каждый скрипт.

А это только один пример очень простой задачи.



И если задуматься, идемпотентность нужна нам везде:

* Проверить существование виртуальной машины. В случае скриптов мы рискуем или плодить бесконечное их множество, или скрипт будет падать в самом начале.
* Какие msi пакеты есть на машине? В лучшем случае тут ничего не упадёт, в худшем машина перестанет адекватно работать.
* Надо ли заново качать билд-артефакты? Хорошо, если ваши билды весят десяток мегабайт. А что делать тем, у кого пару гигабайт?

И другие примеры, где выход из ситуации — это раздувать скрипты бесконечными ветвлениями if'ов, которые невозможно будет адекватно отлаживать и невозможно управлять.

Среди прочих важных для нас вещей можно отметить, что Ansible не использует агентов для управления вашими хостами и машинами. На Linux, понятное дело, он ходит по ssh, а для Windows используется WinRM. Отсюда очевидное следствие: Ansible кроссплатформенный. Он поддерживает какое-то фантастическое количество платформ, вплоть до сетевого оборудования.

И последний, но не менее важный — это YAML формат записи конфигов. Все к нему привыкли, он легко читается, и не составляет труда разобраться, что там происходит.

Но не всё так сладко, есть и проблемы:

* Проблема сомнительная: для запуска плейбуков вам всё равно понадобится Linux-машина, даже если вся ваша инфраструктура исключительно Windows. Хотя это не такая большая проблема в современном мире, т.к. на Windows 10 теперь имеется WSL, где можно запустить Ubuntu, под которой гонять плейбуки.
* Иногда плейбуки действительно сложно отлаживать. Ansible написан на python, и последнее, что хочется увидеть, это простыню питоновского стек-трейса на пять экранов. А виной всему может послужить опечатка в названии модуля

**Как это работает?**  
Для начала нам нужна машина c Linux. В терминологии Ansible это называется Control Machine.  
С неё будут запускаться плейбуки, и на ней происходит вся магия.

На этой машине нам понадобятся:

* Python и питоновский пакетный менеджер pip. На многих дистрибутивах есть из коробки, так что тут без сюрпризов.
* Устанавливаем Ansible через pip, как наиболее универсальный способ: pip install ansible
* Добавляем winrm модуль, чтобы ходить на Windows машины: pip install pywinrm[credssp]
* И на машинах, которыми хотим управлять, надо winrm включить, т.к. по умолчанию он выключен. Есть много способов, как это сделать, и все они описаны в документации Ansible. Но простейший — это взять готовый скрипт из репозитория Ansible и запустить его с требуемым вариантом авторизации: ConfigureRemotingForAnsible.ps1 -EnableCredSSP

Самая важная часть, которую нам надо было получить, чтобы перестать страдать с ps-скриптами — это Inventory. YAML файл (в нашем случае), в котором описана наша инфраструктура и куда можно всегда заглянуть, чтобы понять, где что деплоится. И, конечно же, сами плейбуки. В дальнейшем работа выглядит как запуск плейбука с необходимым инвентори-файлом и дополнительными параметрами.

all:

children:

webservers:

hosts:

spbdotnet-test-host.dev.local:

dbservers:

hosts:

spbdotnet-test-host.dev.local:

vars:

ansible\_connection: winrm

ansible\_winrm\_transport: credssp

ansible\_winrm\_server\_cert\_validation: ignore

ansible\_user: administrator

ansible\_password: 123qweASD

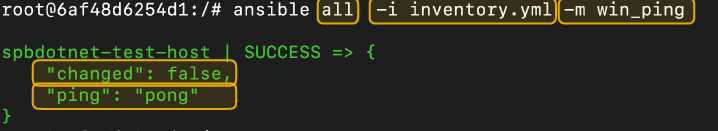
Здесь всё просто: корневая группа all и две подгруппы, webserves и dbservers. Всё остальное интуитивно понятно, только заострю ваше внимание, что Ansible по умолчанию считает, что везде Linux, поэтому для Windows надо обязательно указать winrm и тип авторизации.

Пароль в открытом виде, конечно, в плейбуке хранить не надо, здесь просто пример. Хранить пароли можно, например, в Ansible-Vault. Мы для этого используем TeamCity, который передаёт секреты через переменные окружения и ничего не палит.

**Модули**

Всё что делает Ansible, он делает с помощью модулей. Модули для Linux написаны на python, для Windows на PowerShell. И реверанс в сторону идемпотентности: результат работы модуля всегда приходит в виде json-файла, где указывается, были изменения на хосте или нет.

В общем случае мы будем запускать конструкцию вида ansible группа хостов инвентори файл список модулей:



**Плейбуки**

Плейбук это описание того, как и где мы будем выполнять модули Ansible.

- name: Install AWS CLI

hosts: all

vars:

aws\_cli\_download\_dir: c:\downloads

aws\_cli\_msi\_url: https://s3.amazonaws.com/aws-cli/AWSCLI32PY3.msi

tasks:

- name: Ensure target directory exists

win\_file:

path: "{{ aws\_cli\_download\_dir }}"

state: directory

- name: Download installer

win\_get\_url:

url: "{{ aws\_cli\_msi\_url }}"

dest: "{{ aws\_cli\_download\_dir }}\\awscli.msi"

force: no

- name: Install AWS CLI

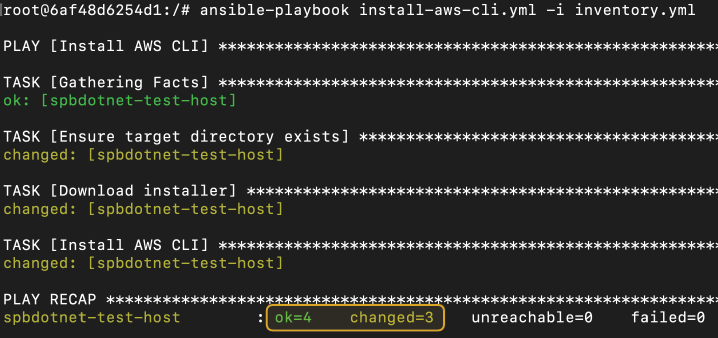
win\_package:

path: "{{ aws\_cli\_download\_dir }}\\awscli.msi"

state: present

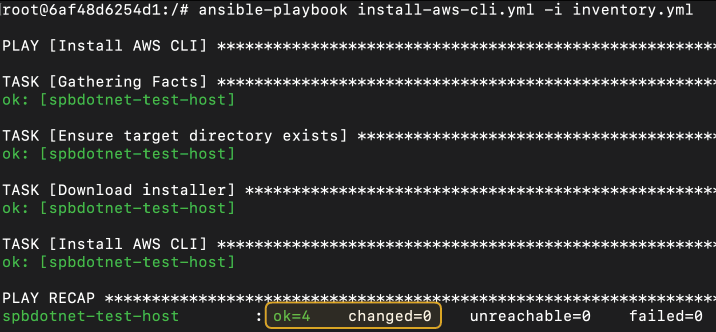
В этом примере у нас три таски. Каждая таска — это вызов модуля. В этом плейбуке мы сначала создаём директорию (убеждаемся, что она есть), затем скачиваем туда AWS CLI и с помощью модуля win\_packge устанавливаем его.

Запустив этот плейбук, мы получим такой результат.



В отчёте видно, что было успешно выполнено четыре таски и три из четырёх произвели какие-то изменения на хосте.

Но что будет, если запустить этот плейбук ещё раз? У нас нигде не написано, что мы должны именно создавать директорию, скачать файл-установщик и запустить его. Мы просто проверяем наличие каждого из пунктов и пропускаем при наличии.



Это и есть та самая идемпотентность, которую мы никак не могли добиться с PowerShell.

**Практика**

Это немного упрощённый пример, но, в принципе, это именно то, что мы делаем каждый день.  
Деплоить будем приложение, состоящее из Windows сервиса и веб приложения под IIS.

- name: Setup App

hosts: webservers

tasks:

- name: Install IIS

win\_feature:

name:

- Web-Server

- Web-Common-Http

include\_sub\_features: True

include\_management\_tools: True

state: present

register: win\_feature

- name: reboot if installing Web-Server feature requires it

win\_reboot:

when: win\_feature.reboot\_required

Для начала нам надо посмотреть, есть ли вообще IIS на хосте, и установить его, если нет. И хорошо бы туда сразу добавить тулзы управления и все зависимые фичи. И совсем хорошо, если хост будет перезагружен при необходимости.

Первую задачу мы решаем модулем win\_feature, который занимается управлением фичами Windows. И тут у нас впервые появляются переменные окружения Ansible, в пункте register. Помните, я говорил, что таски всегда возвращают json объект? Теперь, после выполнения таски Install IIS в переменной win\_feature лежит вывод модуля win\_feature (уж простите за тавтологию).

В следующей таске мы вызываем модуль win\_reboot. Но нам не надо каждый раз перезагружать наш сервер. Мы перезагрузим его, только если модуль win\_feature вернет нам это требование в виде переменной.

Следующим этапом устанавливаем SQL. Чтобы сделать это, придуман уже миллион способов. Я здесь использую модуль win\_chocolatey. Это пакетный менеджер для Windows. Да, именно то самое, к чему мы так привыкли на Linux. Модули поддерживаются комьюнити, и сейчас их уже больше шести тысяч. Очень советую попробовать.

- name: SQL Server

hosts: dbservers

tasks:

- name: Install MS SQL Server 2014

win\_chocolatey:

name: mssqlserver2014express

state: present

Так, хост мы подготовили к запуску приложения, давайте его деплоить!

- name: Deploy binaries

hosts: webservers

vars:

myapp\_artifacts: files/MyAppService.zip

myapp\_workdir: C:\myapp

tasks:

- name: Remove Service if exists

win\_service:

name: MyAppService

state: absent

path: "{{ myapp\_workdir }}\\MyAppService.exe"

На всякий случай, первым делом мы удаляем существующий сервис.

- name: Delete old files

win\_file:

path: "{{ myapp\_workdir }}\\"

state: absent

- name: Copy artifacts to remote machine

win\_copy:

src: "{{ myapp\_artifacts }}"

dest: "{{ myapp\_workdir }}\\"

- name: Unzip build artifacts

win\_unzip:

src: "{{ myapp\_workdir }}\\MyAppService.zip"

dest: "{{ myapp\_workdir }}"

Следующим шагом мы загружаем на хост новые артефакты. В этом плейбуке подразумевается, что запускается он на билд-сервере, все архивы лежат в известной папочке, а путь к ним мы указываем переменными. После копирования (win\_copy) архивы распаковываются(win\_unzip). Дальше мы просто регистрируем сервис, говорим путь к exe и что он должен быть запущен.

- name: Register and start the service

win\_service:

name: ReporterService

start\_mode: auto

state: started

path: "{{ myapp\_workdir }}\\MyAppService.exe"

**Готово!?**

Вроде наш сервис готов к труду и обороне, правда, есть одно "но" — мы не соблюли принцип идемпотентности. Мы всегда удаляем существующий код и потом деплоим новый.

И это проблема. Если мы удалили старый задеплоеный сервис, а после произошла какая-то ошибка, и плейбук не завершил свою работу, мы получим разломанный хост. Или, например, мы деплоим одновременно несколько приложений, из которых одно не изменилось, то нам не надо деплоить его тоже.

Что можно сделать? Как вариант, можно проверять контрольную сумму наших артефактов и сравнивать их с лежащими на сервере.

- name: Get arifacts checksum

stat:

path: "{{ myapp\_artifacts }}"

delegate\_to: localhost

register: myapp\_artifacts\_stat

- name: Get remote artifacts checksum

win\_stat:

path: "{{ myapp\_workdir }}\\MyAppService.zip"

register: myapp\_remote\_artifacts\_stat

Мы используем модуль stat, который предоставляет всякую информацию о файлах и в том числе контрольную сумму. Далее с помощью уже знакомой директивы register пишем результат в переменную. Из интересного: delegate\_to указывает, что это надо выполнить на локальной машине, где запускается плейбук.

- name: Stop play if checksums match

meta: end\_play

when:

- myapp\_artifacts\_stat.stat.checksum is defined

- myapp\_remote\_artifacts\_stat.stat.checksum is defined

- myapp\_artifacts\_stat.stat.checksum == myapp\_remote\_artifacts\_stat.stat.checksum

И с помощью модуля meta мы говорим, что надо закончить выполнение плейбука, если контрольные суммы артефактов на локальной и удалённой машине совпадают. Вот так мы соблюли принцип идемпотентности.

- name: Ensure that the WebApp application exists

win\_iis\_webapplication:

name: WebApp

physical\_path: c:\webapp

site: Default Web Site

state: present

Теперь посмотрим на наше веб-приложение. Опускаем часть про копирование файлов, переходим сразу к сути. Наш билд-сервер сделал паблиш, загружаем всю рассыпуху файлов на хост и используем встроенный модуль для работы с IIS-приложениями. Он создаст приложение и запустит его.

**Переиспользование кода**

Одна из поставленных нами задач была: дать возможность любому инженеру в компании легко запустить деплой. Он пишет свой плейбук из готовых модулей, говорит, что ему надо запустить такой-то продукт на таком-то хосте.

Для этого в Ansible есть Роли. По сути, это конвенция. Мы создаём на сервере папочку /roles/ и кладём в неё наши роли. Каждая роль — это набор конфигурационных файлов: описание наших тасок, переменных, служебных файлов и т.д. Обычно ролью делают какую-то изолированную сущность. Установка IIS — отличный пример, если нам надо не просто его установить, но и как-то дополнительно сконфигурировать или проверить дополнительными тасками. Мы делаем отдельную роль и, таким образом, изолируем все относящиеся к IIS плейбуки в папке с ролями. В дальнейшем мы просто вызываем эту роль директивной include\_role %role\_name%.

Естественно, мы сделали роли для всех приложений, оставив возможность для инженеров, с помощью конфигурационных параметров как-то кастомизировать процесс.

- name: Run App

hosts: webservers

tasks:

- name: "Install IIS"

include\_role:

name: IIS

- name: Run My App

include\_role:

name: MyAppService

vars:

myapp\_artifacts: ./buld.zip

В этом примере для роли Run My App заложена возможность передавать какой-то свой путь к артефактам.

Тут надо замолвить слово про Ansible Galaxy — репозиторий общедоступных типовых решений. Как заведено в приличном обществе, множество вопросов уже было решено до нас. И если возникает ощущение, что вот сейчас мы начнём изобретать велосипед, то сначала надо посмотреть список встроенных модулей, а потом покопаться в Ansible Galaxy. Вполне вероятно, что нужный вам плейбук уже был сделан кем-то другим. Модулей там лежит огромное количество, на все случаи жизни.

**Больше гибкости**

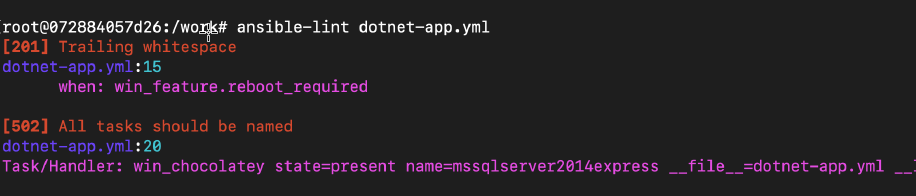
А что делать, если не нашлось ни встроенного модуля, ни подходящей роли в Galaxy? Тут два варианта: или мы что-то делаем не так, или у нас действительно уникальная задача.

В случае второго варианта мы всегда можем написать свой модуль. Как я показывал в начале, Ansible даёт возможность написать простейший модуль буквально за 10 минут, а когда вы углубитесь, вам на помощь приходит довольно подробная документация, покрывающая множество вопросов.

**CI**  
В нашем отделе мы очень любим TeamCity, но тут может быть любой другой CI-сервер на ваш выбор. Для чего нам совместное их использование?

Во-первых, мы всегда можем проверить корректность синтаксиса наших плейбуков. Пока YAML считает табы синтаксической ошибкой, это очень полезная фича.

Также на CI-сервере мы запускаем ansible-lint. Это статический анализатор ansible конфигов, который выдаёт список рекомендаций.



Например, здесь он говорит, что у нас лишний пробел в конце строки, и для одной таски не задано имя. Это важно, т.к. имя модуля может встречаться в одном плейбуке несколько раз, и все таски должны быть именованы.

Само собой, можно ещё писать тесты для плейбуков. Мы можем себе позволить не делать этого, т.к. мы деплоим в тестовую среду, и ничего критичного не случится. А вот если вам деплоить на прод, то лучше бы всё проверить. Благо ansible позволяет тестировать не только плейбуки, но и отдельные модули. Так что обязательно уделите ему внимание.

И вторая основная причина использовать CI-сервер — запуск плейбуков. Это и есть та самая волшебная кнопка "Сделать хорошо", которую даёт нам TeamCity. Мы просто создаём несколько простых конфигураций для разных продуктов, где говорим: ansible-playbook reporter\_vm.yml -i inventory.yml -vvvv и получаем кнопку Deploy.

Бонусная удобность: можно выстраивать зависимости от билдов. Как только что-то сбилдилось, TeamCity запускает процесс редеплоя, после которого нам остаётся только глянуть логи, если вдруг что-то сломалось.

**Итого**

* Запутанные и разрозненные PowerShell скрипты мы заменили YAML-конфигами.
* Разные реализации одинаковых проблем мы заменили общими ролями, которые можно переиспользовать. Создан репозиторий, где лежат роли. Если роль вам подходит, вы просто её используете. Если она вам не подходит, вы просто присылаете пул-реквест, и она вам подходит =)
* Проверить успешность деплоя теперь можно в едином месте.
* Все знают, где искать логи
* Проблемы с коммуникациями тоже решились за счёт общего репозитория и TeamCity. Все заинтересованные люди знают, где лежат плейбуки и как они работают.

P.S. Все примеры из статьи можно взять на [гитхабе](https://github.com/kot-lex/spbdotnet-ansible).

**Теги:**