goodplay Documentation

Release 0.12.0

Benjamin Schwarze

Nov 19, 2018

User Documentation

goodplay enables you to test your deployments and distributed software infrastructure by reusing your existing knowledge of Ansible.

This part of the documentation, which is mostly prose, begins with some background information about goodplay, then focuses on step-by-step instructions for digging deeper into what can be accomplished with goodplay.

CHAPTER 1

Introduction

Writing good tests for your existing deployments or distributed software infrastructure should be painless, and easily accomplishable without involving any time-consuming and complex testing setup. This is where goodplay comes into play.

goodplay instruments Ansible — "a radically simple IT automation platform" as it is advertized — and allows you to write your tests in the same simple and probably already familiar language you would write an Ansible playbook.

1.1 Features

- define your test environments via Docker Compose and Ansible inventories
- write your tests as Ansible 2.x playbook tasks
- resolve and auto-install Ansible role dependencies prior to test run
- run your tests within Docker container(s), an already existing test environment, or on localhost
- built as a pytest plugin to have a solid test runner foundation, plus you can run your goodplay tests together with your other tests

1.2 Versioning

goodplay will use Semantic Versioning when reaching v1.0.0. Until then, the minor version is used for backwards-incompatible changes.

1.3 goodplay vs. Other Software

In this section we compare goodplay to some of the other software options that are available to partly solve what goodplay can accomplish for you.

1.3.1 Ansible

Ansible itself comes bundled with some testing facilities mentioned in the Ansible Testing Strategies documentation. It makes a low-level `assert` module available which helps to verify that some condition holds true, e.g. some output from a previous task which has been stored in a variable contains an expected value.

Although it can be sometimes necessary to use something low-level as Ansible's `assert`, goodplay enables you to use high-level modules for describing your test cases.

Besides the actual testing, goodplay takes care of setting up and tearing down the test environment as well as collecting the test results – both being something Ansible was not made for.

1.3.2 pytest-ansible

pytest-ansible is as the name already implies a pytest plugin just like goodplay. But instead of being used for testing Ansible playbooks or roles, it provides pytest fixtures that allow you to execute Ansible modules from your Python-based tests.

1.3.3 serverspec

serverspec seems to be more targeted to assert hosts are in a defined state. In comparison to goodplay it allows you to run tests against single hosts only and does not include test environment management.

1.4 License

goodplay is open source software released under the Apache License 2.0:

CHAPTER 2

Quick Start

Eager to get started? This page gives a good introduction in how to get started with goodplay.

For our basic example we assume we want to test our existing Ansible playbook that is responsible for installing a plain nginx web server on Ubuntu:

```
## nginx_install.yml
- hosts: web
  tasks:
    - name: install nginx package
      apt:
        name: nginx
        state: latest
        update_cache: yes
```

Briefly summarized, when running the playbook via `ansible-playbook`, Ansible will:

1. Connect to host `web`.
2. Update the cache of `apt`, which is Ubuntu's default package manager.
3. Install the latest `nginx` – one of the most used web servers – package via `apt`.

At a first glance this looks fine but it is not clear if the following holds true:

1. The nginx service is automatically started after the installation.
2. The nginx service is started at boot time.
3. The nginx service is running on port 80.

Let's turn these assumptions into requirements which we are going to test with goodplay. But, first things first... we need to install goodplay.

2.1 Installing

Before installing goodplay make sure you have Docker installed, which is a prerequisite for this quick start tutorial. Check out the official Install Docker Engine guide.

Afterwards, to install goodplay with pip, just run this in your terminal:

```
$ pip install goodplay
```

Please consult the *Installation Guide* for detailed information and alternative installation options.

2.2 Defining Environment

Before writing the actual tests we need to define our test environment which is created as Docker containers behind the scenes. This is done via a Docker Compose file and an Ansible inventory where we define all hosts and groups required for the test run.

In our case we want to test our nginx installation on a single host with Ubuntu Trusty:

```
## tests/docker-compose.yml
version: "2"
services:
  web:
    image: "ubuntu-upstart:trusty"
    tty: True

## tests/inventory
web ansible_user=root
```

In this example we define a *host* (in Docker Compose terminology this is a *service*) with name `web` that runs the official Docker Ubuntu image `ubuntu-upstart:trusty`.

- *Feature: Defining Environment*

2.3 Writing Tests

Now, let's write some tests that ensure nginx is installed according to our requirements:

```
## tests/test_nginx_install.yml
- include: ../nginx_install.yml

- hosts: web
  tasks:
    - name: nginx service is running
      service:
        name: nginx
        state: started
      tags: test

    - name: nginx service is enabled
      service:
        name: nginx
        enabled: yes
```

(continues on next page)

(continued from previous page)

```
    tags: test

  - name: nginx service is listening on port 80
    wait_for:
      port: 80
      timeout: 10
    tags: test
```

You may have noticed that all we have to do is use the same Ansible modules we're already used to. In case you are new to all this playbook stuff, the official Ansible playbook guide will help you getting started.

Labeling a playbook's task with a `test` tag makes goodplay recognize it as a *test task*. A *test task* is meant to be successful (passes) when it does not result in a change and does not fail.

- *Feature: Writing Tests*

2.4 Running Tests

Note: First-time run may take some more seconds or minutes (depending on your internet connection) as the required Docker images need to be downloaded.

The following command will kick-off the test run:

```
$ goodplay -v
============================= test session starts ==============================
platform darwin -- Python 2.7.6, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- /Users
/benjixx/.virtualenvs/goodplay/bin/python2.7
rootdir: /Users/benjixx/src/goodplay/examples/quickstart
plugins: goodplay-0.6.0
collected 3 items

tests/test_nginx_install.yml::nginx service is running PASSED
tests/test_nginx_install.yml::nginx service is enabled PASSED
tests/test_nginx_install.yml::nginx service is listening on port 80 PASSED

========================== 3 passed in 43.13 seconds ===========================
```

CHAPTER 3

Installation

This part of the documentation covers the installation of goodplay.

3.1 Installing Docker

goodplay makes use of isolated containerized environments provided by Docker for running your tests.

Note: If you only require your tests to be run on localhost or some other test environment you manage on your own, you can skip Docker installation and continue with the next section.

As goodplay uses Docker Compose which enables you to use some great Docker features like user-defined networks or embedded DNS server, we recommend to run at least Docker version `1.10.0`. There are a lot of options when it comes to setting up a Docker host.

When running a Linux distribution with a recent kernel version, `docker` is most likely supported natively. In this case the installation process will finish in a minute.

When running on Mac OS X, `docker` is not natively supported (yet). Fortunately there is `docker-machine` available which lets you create Docker hosts as virtual machine on your computer, on cloud providers, or inside your own data center. In this case Docker Toolbox helps you to setup everything you need.

Please make sure to read the official Install Docker Engine guide.

3.2 Installing goodplay

Installing latest released goodplay version is simple with pip, just run this in your terminal:

```
$ pip install goodplay
```

Alternatively you can install the latest goodplay development version:

```
$ pip install git+https://github.com/goodplay/goodplay.git#egg=goodplay
```

3.3 Get the Code

goodplay is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone https://github.com/goodplay/goodplay.git
```

Download the tarball:

```
$ curl -OL https://github.com/goodplay/goodplay/archive/master.tar.gz
```

Or, download the zipball:

```
$ curl -OL https://github.com/goodplay/goodplay/archive/master.zip
```

Once you have a copy of the source, you can install it into your site-packages easily:

```
$ python setup.py install
```

CHAPTER 4

Frequently Asked Questions

4.1 Is Docker required for running goodplay?

Although most people may use goodplay with Docker, it is absolutely fine to run goodplay without Docker and instead run on localhost or against remote hosts. Just keep in mind that you need to take care on your own for setting up and cleaning up your test environment in this case.

4.2 When is a test marked as passed, skipped, or failed?

An executed test always results in one of the following three test outcomes: `passed`, `skipped`, and `failed`. The following table shows the relation of Ansible *task results* of *non-test tasks* and *test tasks* to the actual *test result*.

task result	non-test task	test task
ok	n/a	`passed`
ok (changed)	n/a	`failed`
failed	`global failed`	`failed`
failed (ignore failed)	n/a	n/a
skipped	n/a	`skipped`
unreachable host	`global failed`	`failed`
no hosts	n/a	n/a

These test results are collected for each host a task runs on. At the end of a test task the results are combined to the final test outcome according to the following rules in order:

1. If the task has been failed on one or more hosts test outcome is `failed`.
2. If the task has been skipped on one or more hosts test outcome is `skipped`.
3. Otherwise result in `passed`.

Note:

- In case of a `global failed` this results in a failure with all subsequent tests being `skipped`.
- If all test tasks of a playbook are `skipped` this results in a failure.

4.3 Are test tasks free of side effects?

It depends. Test tasks are run in *check mode* (and thus without side effects) when supported by a module. If *check mode* is not supported, a module is run in normal mode which can result in side effects (depending on a module's functionality).

4.4 My shell/command test always fails. Why?

Since Ansible cannot know when a shell command has changed something, the shell/command task always sets *changed* to *true*. This conflicts with goodplay's assumption, that a task fails if it changed something. To circumvent this, you need to tell Ansible that the shell command did not change using *changed_when*, for example:

```
- name: "check java version"
  shell: java -version 2>&1 | grep -q '1.8.0_122'
  changed_when: False
  tags: test
```

All features provided by goodplay are documented in this section. So if you're trying to use a specific feature you should find all the details here.

CHAPTER 5

Defining Environment

Prior to writing tests it is important to define the environment the tests are going to ran on, e.g. hostnames and platforms. Throughout this documentation we will often refer to this as *inventory*.

goodplay borrows this term from Ansible which already provides various ways to define inventories. When doing a test run, goodplay reads an inventory during setup phase that defines the hosts to be used for the test. These can be hosts you have already available in your environment or Docker containers you have defined via Docker Compose that are automatically created, as we will see in a minute.

The usual and easiest way to define an *inventory* is to create a file named `inventory` right beside the *test playbook*:

```
## inventory
web ansible_user=root
db ansible_user=root
```

This example defines two hosts – `web` and `db`. The remote user that is used to connect to the host needs to be specified via `ansible_user` inventory variable.

5.1 Single Docker Environment

If we would use the inventory example from the previous section together with a test playbook it would not create any Docker containers yet, and thus Ansible would not be able to connect to the hosts `web` and `db`. There are multiple reasons this is not done automatically:

1. goodplay can be used without Docker, e.g. tests can run against localhost or otherwise managed test environment.
2. Some hostnames defined in the inventory may be used only for configuration purposes (not actually required for test run).
3. Hosts may require different platforms, so these must be specified explicitly.

The Docker container environment required for a test run is specified with the help of Docker Compose in a `docker-compose.yml` file right beside the test playbook and inventory.

Note: Please note that Docker Compose uses the term *service* for what goodplay uses the term *host*.

Let's assume we want hosts `web` and `db` to run latest CentOS 7. Therefor we create the following `docker-compose.yml` file:

```
## docker-compose.yml
version: "2"
services:
  web:
    image: "centos:centos7"
    tty: True

  db:
    image: "centos:centos7"
    tty: True
```

When executing a test, goodplay ...

- ... recognizes the `docker-compose.yml` file right beside the test playbook and inventory,
- ... starts up the test environment,
- ... connects the Ansible inventory with the instantiated Docker containers,
- ... executes the test playbook,
- ... and finally shuts down the test environment.

5.2 Multiple Docker Environments

Sometimes you want to run the same test playbook against multiple environments. For example when you have an Ansible role that should support more than one platform, you most likely want to test run it against each supported platform.

We could extend our previous example by not only testing against latest CentOS 7, but also against Ubuntu Trusty:

```
## docker-compose.centos.7.yml
version: "2"
services:
  web:
    image: "centos:centos7"
    tty: True

  db:
    image: "centos:centos7"
    tty: True


## docker-compose.ubuntu.trusty.yml
version: "2"
services:
  web:
    image: "ubuntu-upstart:trusty"
    tty: True

  db:
```

(continues on next page)

(continued from previous page)

```
    image: "ubuntu-upstart:trusty"
    tty: True
```

goodplay will recognize that there are multiple Docker Compose files, and will run the test playbook against each of these environments.

Docker Compose allows you to work with Multiple Docker Compose files. goodplay takes this one step further by introducing conventions to extending/overriding Docker Compose files.

goodplay sees your `docker-compose.yml` files as a hierarchy where as `docker-compose.yml` is the parent of `docker-compose.item1.yml` which is the parent of `docker-compose.item1.item11.yml` and so on and so forth. When deciding which ones to use, goodplay only instantiates the leaves in the hierarchy. Thus you could have intermediate Docker Compose files that hold common configuration that can be refered to further down in the hierarchy.

Additionally one can use the extension `.override.yml` instead of `.yml` to make goodplay override (merge) the Docker Compose file from the same or upper level.

CHAPTER 6

Writing Tests

goodplay builds upon *playbooks* – Ansible's configuration, deployment, and orchestration language.

6.1 Ansible Terminology

Quoting from Ansible's documentation:

> At a basic level, playbooks can be used to manage configurations of and deployments to remote machines. At a more advanced level, they can sequence multi-tier rollouts involving rolling updates, and can delegate actions to other hosts, interacting with monitoring servers and load balancers along the way.

A pseudo *playbook* – written as a YAML file – may look like this:

```
## playbook_name.yml
# play #1
- hosts: host1:host2
  tasks:
    # play #1, task #1
    - name: first task name
      module1:
        arg1: value1
        arg2: value2

    # play #1, task #2
    - name: second task name
      module2:
        arg1: value1
        arg2: value2
      tags: specialtag

# play #2
- hosts: host3
  tasks:
    # play #2, task #1
```

(continues on next page)

(continued from previous page)

```
    - name: another task name
      module1:
        arg1: value1
```

Each *playbook* is composed of one or more *plays*.

Each *play* basically defines two things:

- on which *hosts* to run a particular set of *tasks*, and
- what *tasks* to run on each of these *hosts*.

A *task* refers to the invocation of a *module* which can be e.g. something like creating a user, installing a package, or starting a service. Ansible already comes bundled with a large module library.

6.2 Writing Test Playbooks

After we have briefly introduced the basic terminology of the Ansible language, it is now time to define what a *test playbook* looks like in the goodplay context.

A *test playbook* is as the name implies a *playbook* with the following contraints:

1. The filename is prefixed with `test_`.
2. The filename extension is `.yml`.
3. Right beside the *test playbook* a file or directory named `inventory` exists. See *Defining Environment* for details.
4. If you want to test against Docker containers you may optionally put a `docker-compose.yml` file right beside the *test playbook*.
5. The *test playbook* contains or includes at least one task tagged with `test`, also called *test task*.
6. Within a *test playbook* all *test task* names must be unique.

6.2.1 Basic Example

An example test playbook that verifies that two hosts (`host1` and `host2` created as Docker containers, each one running `centos:centos6` platform image) are reachable:

```
## docker-compose.yml
version: "2"
services:
  host1:
    image: "centos:centos6"
    tty: True

  host2:
    image: "centos:centos6"
    tty: True

## inventory
host1 ansible_user=root
host2 ansible_user=root
```

```
## test_ping_hosts.yml
- hosts: host1:host2
  tasks:
    - name: hosts are reachable
      ping:
      tags: test
```

The name of the single test task is `hosts are reachable`. The test task only passes when the task runs successful on both hosts i.e. both hosts are reachable.

6.2.2 Complex Example

A slightly more complicated example making use of more advanced Ansible features, like defining host groups or registering variables and using Ansible's assert module:

```
## install_myapp.yml
- hosts: myapp-hosts
  tasks:
    - name: install myapp
      debug:
        msg: "Do whatever is necessary to install the app"
```

```
## tests/docker-compose.yml
version: "2"
services:
  host1:
    image: "centos:centos6"
    tty: True

  host2:
    image: "centos:centos6"
    tty: True

## tests/inventory
[myapp-hosts]
host1 ansible_user=root
host2 ansible_user=root
```

```
## tests/test_myapp.yml
- include: ../install_myapp.yml

- hosts: myapp-hosts
  tasks:
    - name: config file is only readable by owner
      file:
        path: /etc/myapp/myapp.conf
        mode: 0400
        state: file
      tags: test

    - name: fetch content of myapp.log
      command: cat /var/log/myapp.log
      register: myapp_log
      changed_when: False
```

(continues on next page)

(continued from previous page)

```
    - name: myapp.log contains no errors
      assert:
        that: "'ERROR' not in myapp_log.stdout"
      tags: test
```

6.3 Writing Tests for Ansible Roles

To keep playbooks organized in a consistent manner and make them reusable, Ansible provides the concept of Ansible Roles. An Ansible role is defined as a directory (named after the role) with subdirectories named by convention:

```
role/
  defaults/
  files/
  handlers/
  meta/
  tasks/
  templates/
  vars/
```

When writing tests for your role, goodplay expects another subdirectory by convention:

```
role/
  ...
  tests/
```

By following this convention, goodplay takes care of making the Ansible role available on the Ansible Roles Path, so you can use them directly in your test playbook.

CHAPTER 7

Auto-Installing Dependencies

Ansible comes bundled with `ansible-galaxy`, a tool to install Ansible roles either from central Ansible Galaxy, or e.g. from a version control system.

goodplay uses `ansible-galaxy` under the hood to auto-install dependencies required by your test playbooks. Dependencies are distiguished into two categories – *hard dependencies* and *soft dependencies*.

> **Warning:** Installing Ansible roles that are maintained by a third-party from Ansible Galaxy may come with its own security risks. So please ensure you know what you're doing and/or install your own roles from your own version control system.

7.1 Hard Dependencies

When writing tests for an Ansible role (i.e. under a role's `tests` directory), goodplay ensures all dependent Ansible roles defined in the role's `meta/main.yml` file are automatically installed and made available in the test context.

We refer to this as *hard dependencies* as these are expected to be required for successfully using an Ansible role.

7.2 Soft Dependencies

Soft dependencies refer to dependent Ansible roles that are only required for test execution, e.g. setting up a third party software component we support to integrate with.

Soft dependencies need to be specified as `requirements.yml` files right beside the test playbook that depends on them, and must follow the guidelines outlined in the Ansible Galaxy Requirements File documentation.

CHAPTER 8

Command-Line Options

Additionally to the default `py.test` command-line options, goodplay provides the following options for `goodplay` and `py.test` executables, which can be seen by passing `--help`:

```
goodplay --help
```

8.1 `--use-local-roles`

By default goodplay creates a temporary directory for installing dependent roles and ensures that has highest precedence when resolving Ansible roles. This is done to ensure your test run doesn't interfere with other roles in your Ansible roles path.

There might be cases where you want to disable this default behavior, and give the configured Ansible roles path highest precedence, e.g.:

1. When you're developing multiple Ansible roles at once and you want to test-run them together.
2. When you cannot use Ansible Galaxy's dependency resolution due to Ansible roles being stored in a non-supported location, e.g. non-supported version control system.

When running with `--use-local-roles` switch, please ensure you have either `ANSIBLE_ROLES_PATH` environment variable set, or `roles_path` configured in your `ansible.cfg`.

8.2 Debugging output

As mentioned in the beginning, goodplay supports `py.test` command-line options. To see the details output of all Ansible tasks you can pass `-v` and `-s` to goodplay:

```
goodplay -v -s
```

CHAPTER 9

Integrating with Third Parties

9.1 GitLab CI

GitLab CI is part of GitLab. You can use it for free on GitLab.com.

```
## .gitlab-ci.yml
image: goodplay/goodplay

services:
  - docker:dind

test:
  script:
    - goodplay -v -s
```

9.2 Travis CI

Travis CI is a continuous integration service that is available to open source projects at no cost.

```
## .travis.yml
sudo: required
dist: trusty

language: python
python: 2.7

services:
  - docker

before_install:
  # ensure apt-get cache is up-to-date
```

(continues on next page)

(continued from previous page)

```
  - sudo apt-get -qq update

  # upgrade docker-engine to latest version
  - export DEBIAN_FRONTEND=noninteractive
  - sudo apt-get -qq -o Dpkg::Options::="--force-confnew" -y install docker-engine
  - docker version

install:
  - pip install goodplay

script:
  - goodplay -v
```

9.3 Jenkins CI

To run on Jenkins CI you have to configure the following in your build job:

1. Under section **Build** choose **Add build step > Execute shell** with

   ```
   pip install goodplay
   goodplay -v --junit-xml=junit.xml
   ```

2. Under section **Post-build Actions** choose **Add post-build action > Publish JUnit test result report** and set **Test report XMLs** to `**/junit.xml`.

9.4 pytest

goodplay is built as a pytest plugin which is enabled by default. Thus when running your other tests via `py.test` command-line interface, pytest also runs the goodplay tests right beside them.

Note: When running `goodplay` command-line interface only goodplay tests are considered.

In case you need some inspiration you should have a look at our real-world examples that showcase how goodplay is used in the wild.

CHAPTER 10

What are you doing with goodplay?

Note: This is reserved for your real-world examples. Please feel free to add your *project name* and *project link* to the list.

- None yet. Why not be the first?

If you want to contribute to the project, this part of the documentation is for you.

CHAPTER 11

Contributor's Guide

If you're reading this you're probably interested in contributing to goodplay. First, we'd like to say: thank you! Open source projects live-and-die based on the support they receive from others, and the fact that you're even considering supporting goodplay is very generous of you.

This document lays out guidelines and advice for contributing to goodplay. If you're thinking of contributing, start by reading this thoroughly and getting a feel for how contributing to the project works.

The guide is split into sections based on the type of contribution you're thinking of making, with a section that covers general guidelines for all contributors.

11.1 All Contributions

11.1.1 Get Early Feedback

If you are contributing, do not feel the need to sit on your contribution until it is perfectly polished and complete. It helps everyone involved for you to seek feedback as early as you possibly can. Submitting an early, unfinished version of your contribution for feedback in no way prejudices your chances of getting that contribution accepted, and can save you from putting a lot of work into a contribution that is not suitable for the project.

11.1.2 Contribution Suitability

The project maintainer has the last word on whether or not a contribution is suitable for goodplay. All contributions will be considered, but from time to time contributions will be rejected because they do not suit the project.

If your contribution is rejected, don't despair! So long as you followed these guidelines, you'll have a much better chance of getting your next contribution accepted.

11.2 Code Contributions

11.2.1 Steps

When contributing code, you'll want to follow this checklist:

1. Fork the repository on GitHub.
2. Run the tests to confirm they all pass on your system. If they don't, you'll need to investigate why they fail. If you're unable to diagnose this yourself, raise it as a bug report by following the guidelines in this document: *Bug Reports*.
3. Write tests that demonstrate your bug or feature. Ensure that they fail.
4. Make your change.
5. Run the entire test suite again, confirming that all tests pass *including the ones you just added*.
6. Send a GitHub Pull Request to the main repository's `master` branch. GitHub Pull Requests are the expected method of code collaboration on this project.

11.2.2 Code Review

Contributions will not be merged until they've been code reviewed. You should implement any code review feedback unless you strongly object to it. In the event that you object to the code review feedback, you should make your case clearly and calmly. If, after doing so, the feedback is judged to still apply, you must either apply the feedback or withdraw your contribution.

11.3 Documentation Contributions

Documentation improvements are always welcome! The documentation files live in the `docs/` directory of the codebase. They're written in reStructuredText, and use Sphinx to generate the full suite of documentation.

When contributing documentation, please attempt to follow the style of the documentation files. This means a soft-limit of 79 characters wide in your text files and a semi-formal prose style.

11.4 Bug Reports

Bug reports are hugely important! Before you raise one, though, please check through the GitHub issues, **both open and closed**, to confirm that the bug hasn't been reported before. Duplicate bug reports are a huge drain on the time of other contributors, and should be avoided as much as possible.

11.5 Feature Requests

When you're missing some feature, feel free to raise a feature request through the GitHub issues. Please ensure beforehand that the same feature request doesn't exist yet.

CHAPTER 12

Authors

12.1 Development Lead

- Benjamin Schwarze <benjamin.schwarze@mailboxd.de> (@benjixx)

12.2 Contributors

- Eric Van Steenbergen <vs.eric@gmail.com>
- Sebastian May <me@bstr.eu>

12.3 Credits

Special thanks goes to the requests project which heavily inspired our contribution guidelines.

CHAPTER 13

History

13.1 0.12.0 (2018-11-19)

13.1.1 Major Changes

- add support for Ansible 2.6 and 2.7, drop support for Ansible 2.3 and 2.4
- add support for Python 3.7

13.1.2 Minor Changes

- fix docker-compose integration after method signature change in `docker-compose==1.23.0`
- update dependencies to newer versions

13.2 0.11.0 (2018-06-20)

13.2.1 Minor Changes

- update dependencies to newer versions

13.3 0.10.0 (2018-03-26)

13.3.1 Major Changes

- add support for Ansible 2.5, drop support for Ansible 2.2
- require `pytest>=3.5.0` due to a change in their nodeid calculation

13.4 0.9.1 (2018-01-15)

13.4.1 Minor Changes

- report appropriate build error message when building from docker-compose
- fix warning "Module already imported so cannot be rewritten: goodplay"

13.5 0.9.0 (2017-12-25)

13.5.1 Minor Changes

- when using docker-compose.yml files in tests with referenced Dockerfiles, a build is triggered before bringing up the containers (NOT attempting to pull the latest base image as image might be only available locally)

13.6 0.8.1 (2017-12-19)

13.6.1 Minor Changes

- require `docker-compose>=1.18.0` due to a method signature change
- when using docker-compose.yml files in tests with referenced Dockerfiles, a build is triggered before bringing up the containers (always attempting to pull the latest base image)

13.7 0.8.0 (2017-10-15)

13.7.1 Major Changes

- add support for Ansible 2.2, 2.3, and 2.4, drop support for Ansible 2.1
- add support for Docker 1.12 and greater, drop support for Docker 1.11 and below
- add support for Python 3.6, now effectively supporting Python 2.7 and 3.6
- update to pytest 3
- provide Docker image `goodplay/goodplay`

13.7.2 Minor Changes

- mention GitLab CI support in the docs

13.7.3 Internal Changes

- improve Python-Ansible combinations that are tested on Travis CI

13.8 0.7.0 (2016-06-18)

13.8.1 Major Changes

- support `become_user` with Docker's native user management when running privilege escalation task against Docker Compose environment thus `sudo` is not required in a Docker container anymore; this may change in a future version once Ansible supports `su` with Docker connection plugin
- drop support for `ansible==2.0.x`, now require `ansible>=2.1.0`

13.8.2 Bug Fixes

- fix issue with using local Ansible roles (`--use-local-roles`)
- fix wait_for test task that timeouts or otherwise fails resulting in global fail

13.8.3 Internal Changes

- skip Docker-related tests when Docker is not available
- run Travis CI tests against latest two Docker minor versions, each with latest patch version
- add tests for automatic check mode usage when using a custom module that supports check mode

13.9 0.6.0 (2016-04-28)

13.9.1 Major Changes

- use Docker Compose for defining environments instead of reinventing the wheel, thus bringing you all the latest and greatest features of Docker Compose (e.g. running from Dockerfile, custom networks, custom entrypoints, shared volumes, and more)
- support running any test playbook (not only Ansible role playbooks) against multiple environments
- test tasks now run in check mode when supported by module
- remove `goodplay_image` and `goodplay_platform` support from inventory files
- remove `.goodplay.yml` support as it has only been used for defining platform-name-to-docker-image mapping which is now handled by Docker Compose

13.9.2 Minor Changes

- now depend on `pytest>=2.9.1,<3`

13.9.3 Other Improvements

- fresh goodplay logo
- do not display traceback for goodplay failures

13.10 0.5.0 (2016-03-20)

13.10.1 Major Changes

- goodplay now requires at least Docker 1.10.0
- docker: make use of user-defined networks to isolate test environments
- docker: hosts can now resolve each other thanks to Docker's embedded DNS server
- support use of local Ansible roles (`--use-local-roles`) during test run

13.10.2 Bug Fixes

- add missing `ansible_user` inventory variable in tests as this is required for latest Docker connection plugin in Ansible
- fix junitxml support for `pytest>=2.9.1`

13.10.3 Other Improvements

- ease test writing by introducing `smart_create` helper
- speed-up tests by using `gather_facts: no` where possible
- docs: compare goodplay to other software
- add gitter chat badge
- explicitly disable Ansible retry files

13.11 0.4.1 (2016-01-22)

13.11.1 Major Changes

- repository moved to new organization on GitHub: goodplay/goodplay

13.11.2 Bug Fixes

- fix host vars getting mixed due to Ansible caches being kept as module state

13.12 0.4.0 (2016-01-13)

13.12.1 Major Changes

- add support for testing against defined Docker environment
- make latest Ansible 2.0 release candidate install automatically
- massive documentation refactorings, now available under https://docs.goodplay.io/
- introduce command line interface: goodplay

- drop Ansible 1.9.x support to move things forward

13.12.2 Bug Fixes

- fix goodplay plugin missing when running Ansible

13.12.3 Internal Changes

- switch from traditional Code Climate to new Code Climate Platform
- disable use_develop in tox.ini to more closely match a real user's environment
- refactor code to have sarge integrated at a single point

13.13 0.3.0 (2015-09-07)

13.13.1 Major Changes

- add support for Ansible role testing
- add support for auto-installing Ansible role dependencies (hard dependencies)
- add support for auto-installing soft dependencies

13.13.2 Bug Fixes

- fix test failing when previous non-test task has been changed
- fix failing non-test task after all completed test tasks not being reported as failure

13.13.3 Internal Changes

- use ansible-playbook subprocess for collecting tests as Ansible does not provide an official Python API and Ansible internals are more likely to be changed
- various code refactorings based on Code Climate recommendations
- switch to Travis CI for testing as it now supports Docker

13.14 0.2.0 (2015-08-24)

- initial implementation of Ansible v1 and v2 test collector and runner

13.15 0.1.0 (2015-07-22)

- first planning release on PyPI