Chapter 9 - Deployments with Ansible

Deploying application code to servers is one of the hardest, but most rewarding, tasks of any development team. Most shops using traditional deployment techniques (manual steps, shell scripts, and prayers) dread deployments, especially for complex, monolithic apps.

Deployments are less daunting when you adopt modern deployment processes and use the right amount of automation. In the best case, deployments become so boring and routine they barely register as a blip on your team's radar.

Consider Etsy, a company whose engineers are deploying code to production up to 40 times per day[100], with no manual intervention from the operations team. The operations team is free to work on more creative endeavors, and the developers see their code go live in near-real-time!

Etsy's production deployment schedule is enabled by a strong DevOps-oriented culture (with robust code repository management, continuous integration, well-tested code, feature flags, etc.). While it may not be immediately possible to start deploying *your* application to production 20 times a day, you can move a long way towards effortless deployments by automating deployments with Ansible.

Deployment strategies

There are dozens of ways to deploy code to servers. For the most basic applications, you may only need to switch to a new tag in a code repository on the server and restarting a service.

For more complex applications, you might do a full Blue-Green deployment, where you build an entire new infrastructure alongside your current production infrastructure, run tests on the new infrastructure, then automatically cut over to the new

[100]http://www.slideshare.net/mikebrittain/principles-and-practices-in-continuous-deployment-at-etsy

instances. This may be overkill for many applications (especially if <100% uptime is acceptable), but it is becoming more and more common—and Ansible automates the entire process.

In this chapter, we will be covering the following deployment strategies:

1. Single-server deployments.
2. Zero-downtime multi-server deployments.
3. Capistrano-style and blue-green deployments.

These are three of the most common deployment techniques, and they cover many common use cases. There are other ways to strengthen your deployment processes, often involving application-level and organizational change, but those deployment aspects are out of the scope of this book.

Simple single-server deployments

The vast majority of small applications and websites are easily run on a single virtual machine or dedicated server. Using Ansible to provision and manage the configuration on the server is a no-brainer. Even though you only have to manage *one* server, it's better to encapsulate all the setup so you don't end up with a *snowflake server*.

In this instance, we are managing a Ruby on Rails site that allows users to perform CRUD operations on articles (database records with a title and body).

The code repository for this app is located on GitHub in the demo-rails-app repository[101].

Begin by creating a new Vagrant VM for local testing using the following Vagrantfile:

[101]https://github.com/geerlingguy/demo-rails-app

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure(2) do |config|
  config.vm.box = "geerlingguy/ubuntu1804"

  config.vm.provider "virtualbox" do |v|
    v.name = "rails-demo"
    v.memory = 1024
    v.cpus = 2
  end

  config.vm.hostname = "rails-demo"
  config.vm.network :private_network, ip: "192.168.33.7"

  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "playbooks/main.yml"
    ansible.become = true
  end

end
```

In this case, we have a VM at the IP address `192.168.33.7`. When provisioned, it will run the Ansible playbook defined in `playbooks/main.yml` on the VM.

Provisioning a Ruby on Rails server

To prepare for our application deployment, we need to do the following:

1. Install git (our application is version controlled in a git repository).
2. Install Node.js (asset compilation requires its Javascript runtime).
3. Install Ruby (our application requires version 2.6.0 or later).
4. Install Passenger with Nginx (we need a fast web server to run our rails application).
5. Install any other dependencies, and prepare the server for deployment.

Let's create a new playbook just for the provisioning tasks (we'll worry about deployment later), in a new file, `playbooks/provision.yml`:

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

  vars_files:
    - vars.yml

  roles:
    - geerlingguy.git
    - geerlingguy.nodejs
    - geerlingguy.ruby
    - geerlingguy.passenger

  tasks:
    - name: Install app dependencies.
      apt:
        name:
          - libsqlite3-dev
          - libreadline-dev
          - tzdata
        state: present

    - name: Ensure app directory exists and is writeable.
      file:
        path: "{{ app_directory }}"
        state: directory
        owner: "{{ app_user }}"
        group: "{{ app_user }}"
        mode: 0755
```

This is a straightforward playbook. We'll need to define a few variables to make sure the `geerlingguy.ruby` role installs the correct version of Ruby (at least 2.6.0), and the `geerlingguy.passenger` role is configured to serve our app correctly.

There are also a few other variables we will need, like `app_directory` and `app_user`, so let's create the variables file now, at `playbooks/vars.yml`:

```
# Variables for our app.
app_directory: /opt/demo-rails-app
app_user: www-data

# Variables for Passenger and Nginx.
passenger_server_name: 0.0.0.0
passenger_app_root: /opt/demo-rails-app/public
passenger_app_env: production
passenger_ruby: /usr/local/bin/ruby

# Variables for Ruby installation.
ruby_install_from_source: true
ruby_download_url: https://cache.ruby-lang.org/pub/ruby/2.6/\
ruby-2.6.0.tar.gz
ruby_version: 2.6.0
```

The passenger variables tell Passenger to run a server available on every network interface, and to launch our app (which will be located in `/opt/demo-rails-app/public`) with `production` settings (the app's environment), using the `ruby` binary we have installed in `/usr/local/bin/ruby`.

The Ruby variables tell the `ruby` role to install Ruby 2.6.0 from source, since the packages available through Ubuntu's standard apt repositories only contain older versions.

The playbook specified in our Vagrantfile, `playbooks/main.yml`, doesn't yet exist. Let's create the playbook and include the above `provisioning.yml` playbook so our server will be provisioned successfully. We'll separate out the deployment steps into another playbook and include that separately. Inside `playbooks/main.yml`:

```
---
- import_playbook: provision.yml
```

Deploying a Rails app to the server

All the dependencies for our app's deployment were configured in `provision.yml`, so we're ready to build a playbook to perform all the deployment tasks.

Add a line to the `main.yml` file to include a new `deploy.yml` playbook:

```
---
- import_playbook: provision.yml
- import_playbook: deploy.yml
```

Now we're ready to create the `deploy.yml` playbook, which will do the following:

1. Use git to check out the latest production release of the Rails app.
2. Copy over a `secrets.yml` template with secure app data required for running the app.
3. Make sure all the gems required for the app are installed (via Bundler).
4. Create the database (if it doesn't already exist).
5. Run `rake` tasks to make sure the database schema is up-to-date and all assets (like JS and CSS) are compiled.
6. Make sure the app files' ownership is set correctly so Passenger and Nginx serve them without error.
7. If any changes or updates were made, restart Passenger and Nginx.

Most of these tasks will use Ansible's modules, but for a few, we'll just wrap the normal deployment-related commands in `command` since there aren't pre-existing modules to take care of them for us:

```
---
- hosts: all
  become: yes

  vars_files:
    - vars.yml

  roles:
    - geerlingguy.passenger

  tasks:
    - name: Ensure demo application is at correct release.
      git:
        repo: https://github.com/geerlingguy/demo-rails-app.git
        version: "{{ app_version }}"
        dest: "{{ app_directory }}"
        accept_hostkey: true
        force: yes
      register: app_updated
      notify: restart nginx

    - name: Ensure secrets file is present.
      template:
        src: templates/secrets.yml.j2
        dest: "{{ app_directory }}/config/secrets.yml"
        owner: "{{ app_user }}"
        group: "{{ app_user }}"
        mode: 0664
      notify: restart nginx

    - name: Install required dependencies with bundler.
      command: "bundle install --path vendor/bundle \
chdir={{ app_directory }}"
      when: app_updated.changed == true
      notify: restart nginx

    - name: Check if database exists.
      stat: "path={{ app_directory }}/db/{{ app_environment.\
RAILS_ENV }}.sqlite3"
      register: app_db_exists

    - name: Create database.
      command: "bundle exec rake db:create chdir={{ app_directory }}"
      when: app_db_exists.stat.exists == false
      notify: restart nginx

    - name: Perform deployment-related rake tasks.
      command: "{{ item }} chdir={{ app_directory }}"
      with_items:
        - bundle exec rake db:migrate
        - bundle exec rake assets:precompile
      environment: "{{ app_environment }}"
      when: app_updated.changed == true
      notify: restart nginx

    - name: Ensure demo application has correct user for files.
      file:
        path: "{{ app_directory }}"
        state: directory
        owner: "{{ app_user }}"
        group: "{{ app_user }}"
        recurse: yes
      notify: restart nginx
```

The first thing you'll notice (besides the fact we've included the `vars.yml` file again, since we need those variables in this playbook as well) is we've added the `geerlingguy.passenger` role in this playbook. Since we'll be using one of the handlers defined in that role (`restart nginx`), we need to include the role explicitly. We could've added a separate handler specific to this playbook, but using a role's handlers usually makes maintenance easier.

Let's walk through the tasks, one-by-one:

1. (Lines 12-20) We put all the application files in place by checking out the git repository at the version `app_version` into the directory `app_directory`. We set `accept_hostkey` to true so this task doesn't hang the first time we deploy the app (since we haven't yet accepted the Git server's hostkey).
2. (Lines 22-29) We copy a `secrets.yml` file to the application's configuration directory. There are different ways to deploy app secrets, but this is the easiest, and allows us to store the app secrets in an Ansible Vault-protected vars file if we so desire.
3. (Lines 31-34) If the `app_updated` variable shows a change occurred as part of the first `git` task, we'll run a bundler command to ensure all the latest bundled dependencies are installed in the `vendor/bundle` directory.
4. (Lines 36-43) Create the application database with `rake db:create` if it doesn't already exist. Since this application uses a SQLite database, it's a matter of checking if the .sqlite3 file exists, and if not, running the `db:create` task.
5. (Lines 45-52) If the `app_updated` variable shows a change occurred as part of the first `git` task, we'll also run a couple `rake` tasks to make sure the database schema is up to date, and all assets (like scripts and stylesheets) are compiled.
6. (Lines 54-61) Make sure all app files have the correct permissions for Passenger/Nginx to serve them correctly.

Because many of the tasks result in filesystem changes that could change the behavior of the application, they all notify the `restart nginx` handler provided by the `geerlingguy.passenger` role, so Passenger reloads the configuration and restarts the app.

There are a few new variables we need to add to `vars.yml`, and we also need to add the `secrets.yml.j2` template mentioned in the task that copies it into place.

First, we'll create the secrets file, inside `playbooks/templates/secrets.yml.j2`:

```
development:
  secret_key_base: {{ app_secrets.dev }}

test:
  secret_key_base: {{ app_secrets.test }}

production:
  secret_key_base: {{ app_secrets.prod }}
```

We'll be using a dictionary variable for `app_secrets`, so let's add it and all the other new variables to `playbooks/vars.yml`:

```
---
# Variables for our app.
app_version: 2.0.1
app_directory: /opt/demo-rails-app
app_user: www-data
app_secrets:
  dev: fe562ec1e21eecc5af4d83f6a157a7
  test: 4408f36dd290766d2f368fdfcedf4d
  prod: 9bf801da1a24c9a103ea86a1438caa
app_environment:
  RAILS_ENV: production

# Variables for Passenger and Nginx.
passenger_server_name: 0.0.0.0
passenger_app_root: /opt/demo-rails-app/public
passenger_app_env: production
passenger_ruby: /usr/local/bin/ruby

# Variables for Ruby installation.
ruby_install_from_source: true
ruby_download_url: http://cache.ruby-lang.org/pub/ruby/2.2/\
ruby-2.2.0.tar.gz
ruby_version: 2.2.0
```

Note the addition of the following variables to support our `deploy.yml` playbook:

- `app_version`: This is the git tag, branch, or commit hash to be deployed to the server.
- `app_secrets`: A dictionary of Rails app secrets, which are used to verify the integrity of signed app cookies. You can generate new, unique strings for these variables using `rake secret`.
- `app_environment`: Environment settings required for certain commands (like `bundle exec` and `rake`) to run with the correct Rails application environment.

Provisioning and Deploying the Rails App

Since we now have our `provision.yml` and `deploy.yml` playbooks completed, and both are `include`d in the `main.yml` playbook Vagrant will run, it's time to bring up the new VM using Vagrant, and see if our application works!

The structure of your project folder should look like this:

```
deployments/
  playbooks/
    templates/
      secrets.yml.j2
    deploy.yml
    main.yml
    provision.yml
    vars.yml
  Vagrantfile
```

Before running the playbook, we need to make sure all the role dependencies are present. If you were building everything from scratch, you might have a `roles` directory with all the roles inside, but in this case, since we're using roles from Ansible Galaxy, it's best to not include the role files directly with our playbook, but instead, add a `requirements.yml` file to the project and install the roles automatically with Galaxy.

Inside `requirements.yml`:

```
---
- src: geerlingguy.git
- src: geerlingguy.ruby
- src: geerlingguy.nodejs
- src: geerlingguy.passenger
```

In the same directory as the requirements file, run the command `$ ansible-galaxy install -r requirements.yml`, and after a minute, all the required roles will be downloaded to your default Ansible roles directory, if they're not already present.

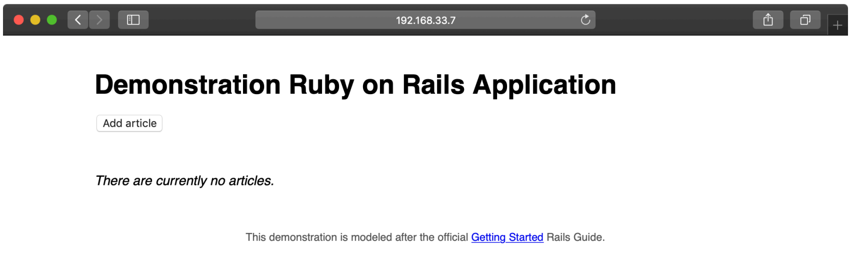
Change directory back to the main directory containing the `Vagrantfile`, and run `vagrant up`. Assuming everything runs correctly, you should see the playbook complete successfully after a few minutes:

```
TASK [Ensure demo application has correct user for files.] **********
changed: [default]

RUNNING HANDLER [geerlingguy.passenger : restart nginx] *************
changed: [default]

PLAY RECAP **********************************************************
default                    : ok=55   changed=31   unreachable=0    failed=0
```

Now, jump over to a web browser and load `http://192.168.33.7/`. You should see something like the following:



Demonstration Ruby on Rails Application

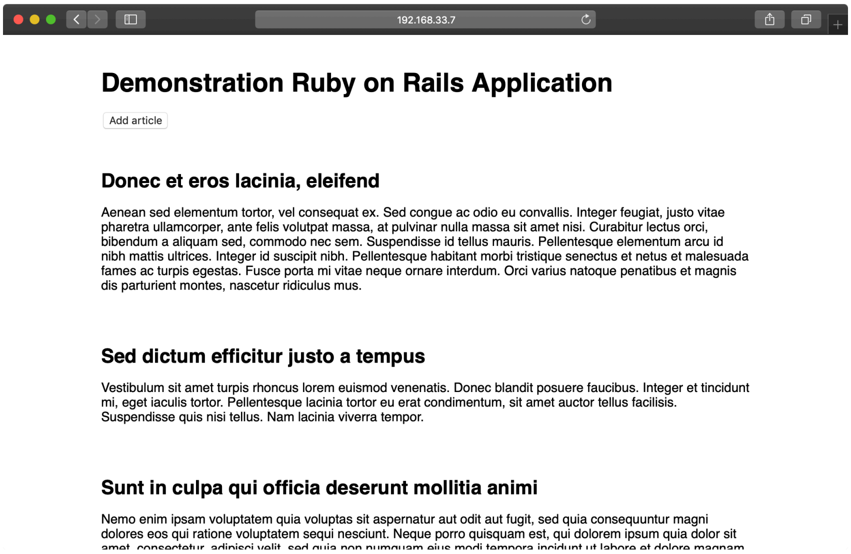
Add article

There are currently no articles.

This demonstration is modeled after the official Getting Started Rails Guide.

Demonstration Rails app running successfully.

Try creating, updating, and deleting a few articles to make sure the database and all app functionality is working correctly:



Demonstration Ruby on Rails Application

Donec et eros lacinia, eleifend

Aenean sed elementum tortor, vel consequat ex. Sed congue ac odio eu convallis. Integer feugiat, justo vitae pharetra ullamcorper, ante felis volutpat massa, at pulvinar nulla massa sit amet nisi. Curabitur lectus orci, bibendum a aliquam sed, commodo nec sem. Suspendisse id tellus mauris. Pellentesque elementum arcu id nibh mattis ultrices. Integer id suscipit nibh. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Fusce porta mi vitae neque ornare interdum. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Sed dictum efficitur justo a tempus

Vestibulum sit amet turpis rhoncus lorem euismod venenatis. Donec blandit posuere faucibus. Integer et tincidunt mi, eget iaculis tortor. Pellentesque lacinia tortor eu erat condimentum, sit amet auctor tellus facilisis. Suspendisse quis nisi tellus. Nam lacinia viverra tempor.

Sunt in culpa qui officia deserunt mollitia animi

Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam

A Rails app to perform CRUD operations on Articles.

The app seems to function perfectly, but it could use some improvements. After more development work, we have a new version of to deploy. We could update the `app_version` variable in `vars.yml` and run `vagrant provision` to run the entire provisioning and deployment playbook again, but to save time using our flexible playbook layout (with separate provisioning and deployment playbooks), we can run the `deploy.yml` playbook separately.

Deploying application updates

First, to test deployment without provisioning, we need to create an inventory file to tell Ansible how to connect directly to the Vagrant-managed VM.

Create the file `playbooks/inventory-ansible` with the following contents:

```
[rails]
192.168.33.7

[rails:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

If you were creating this playbook for a server or VM running outside of Vagrant's control, you'd probably have already created an inventory file or added the server to your global inventory, but when we're working with Vagrant, it's often convenient to use Vagrant's own dynamically-managed inventory. Running playbooks outside of Vagrant's `up/provision` functionality requires us to create a separate inventory file.

Test the ability to run the `deploy.yml` playbook by running the following command inside the `playbooks` directory:

```
$ ansible-playbook deploy.yml -i inventory-ansible
```

Hopefully the playbook completed its run successfully. It may have reported a change in the "Ensure demo application has correct user for files" task, and if so, it will have restarted Passenger. Run it again, and ansible should report no changes:

```
PLAY RECAP *********************************************************
192.168.33.7         : ok=18   changed=0    unreachable=0    failed=0
```

Hopefully you've noticed running the `deploy.yml` playbook standalone is much faster than running the `provision` and `deploy` playbooks together (deployment only takes 18 tasks, while both playbooks add up to 80+ tasks!). In the future, we can deploy application updates using only the `deploy.yml` playbook and changing the `app_version` either in `vars.yml` or by specifying the version on the command line in the `ansible-playbook` command.

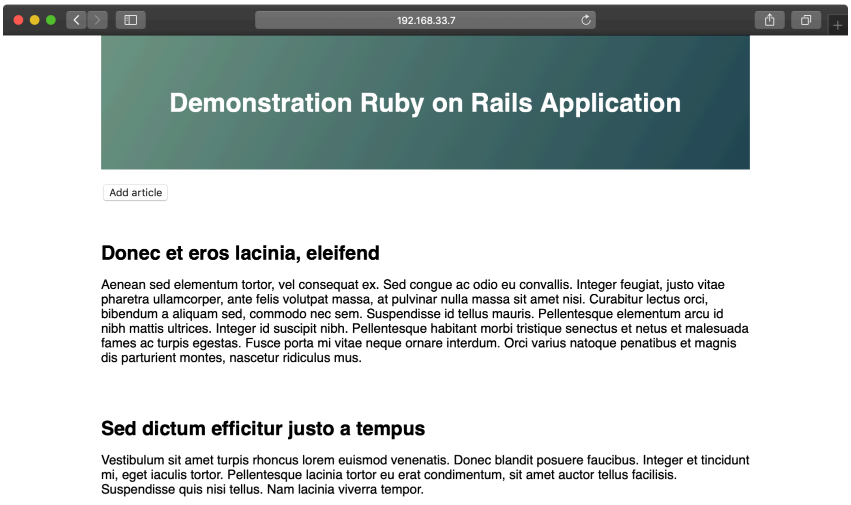
It's generally preferred to change variables in vars files that are versioned with your playbooks, rather than specify them through inventory files, environment variables, or on the command line. This way the entire state of your infrastructure is encapsulated in your playbook files, which ideally should be version controlled and managed similarly to the application they deploy. Plus, who wants to enter any more information on the command line than is absolutely required?

Our application is a fairly generic web application with updates to application code (which require a web server reload), styles (which need recompiling), and possibly the database schema (which needs `rake` migrate tasks to be run). Any time `app_version` is changed inside `playbooks/vars.yml`, the deploy playbook will automatically run all the required tasks to get our app running with the latest code.

Update `app_version` to `2.1.0`, and then run the following command again:

```
$ ansible-playbook deploy.yml -i inventory-ansible
```

After a minute or so, the deployment should complete. Once complete, you'll see the new-and-improved version of the Demonstration Ruby on Rails Application:



Donec et eros lacinia, eleifend

Aenean sed elementum tortor, vel consequat ex. Sed congue ac odio eu convallis. Integer feugiat, justo vitae pharetra ullamcorper, ante felis volutpat massa, at pulvinar nulla massa sit amet nisi. Curabitur lectus orci, bibendum a aliquam sed, commodo nec sem. Suspendisse id tellus mauris. Pellentesque elementum arcu id nibh mattis ultrices. Integer id suscipit nibh. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Fusce porta mi vitae neque ornare interdum. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Sed dictum efficitur justo a tempus

Vestibulum sit amet turpis rhoncus lorem euismod venenatis. Donec blandit posuere faucibus. Integer et tincidunt mi, eget iaculis tortor. Pellentesque lacinia tortor eu erat condimentum, sit amet auctor tellus facilisis. Suspendisse quis nisi tellus. Nam lacinia viverra tempor.

Rails app - version 2.1.0 with a fancy header.

Application update deployments involve incrementing the `app_version` to the latest git tag, then running the `deploy.yml` playbook again. The `main.yml` playbook can be run to ensure the entire server stack is in the correct state, but it's faster to just deploy the app updates.

Code for this entire example is available on GitHub in the Ansible for DevOps deployments example[102].

Zero-downtime multi-server deployments

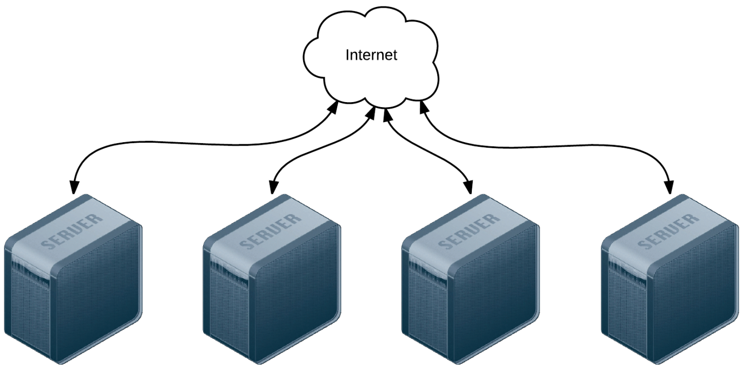
If you need to run an application on multiple servers for horizontal scalability or redundancy, deployments can be cumbersome, resulting in downtime and complicated deployment processes—but not when you use Ansible!

Server Check.in[103] is a server and website monitoring service with a microservices-based architecture; there is a website, an API application, and a server checking application.

The server checking application needs to run on a variety of servers hosted around the world by different providers to provide redundancy and reliability. Server Check.in uses Ansible to manage *rolling deployments* for this application, so new code is deployed across all the servers in minutes while maintaining 100% uptime!

We'll emulate part of Server Check.in's infrastructure (the check server application) by deploying and updating a small Node.js application to a set of virtual machines. The code repository for this app is located on GitHub in the demo-nodejs-api repository[104]. Here's a diagram of the infrastructure we'll be building:

[102]https://github.com/geerlingguy/ansible-for-devops/tree/master/deployments
[103]https://servercheck.in/
[104]https://github.com/geerlingguy/demo-nodejs-api



Four servers connected to the Internet.

To begin, create four lightweight Vagrant VMs using the following Vagrantfile:

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  # Base VM OS configuration.
  config.vm.box = "geerlingguy/ubuntu1804"
  config.vm.synced_folder '.', '/vagrant', disabled: true
  config.ssh.insert_key = false

  config.vm.provider :virtualbox do |v|
    v.memory = 256
    v.cpus = 1
    v.linked_clone = true
  end

  # Define four VMs with static private IP addresses.
  boxes = [
    { :name => "nodejs1", :ip => "192.168.3.2" },
    { :name => "nodejs2", :ip => "192.168.3.3" },
    { :name => "nodejs3", :ip => "192.168.3.4" },
    { :name => "nodejs4", :ip => "192.168.3.5" }
  ]

  # Provision each of the VMs.
  boxes.each do |opts|
    config.vm.define opts[:name] do |config|
      config.vm.hostname = opts[:name]
      config.vm.network :private_network, ip: opts[:ip]

      # Provision all the VMs using Ansible after last VM is up.
      if opts[:name] == "nodejs4"
        config.vm.provision "ansible" do |ansible|
          ansible.playbook = "playbooks/main.yml"
          ansible.inventory_path = "inventory"
          ansible.limit = "all"
        end
      end
    end
  end

end
```

The above `Vagrantfile` defines four VMs with 256MB of RAM and a unique hostname and IP address (defined by the `boxes` variable). Our Node.js app doesn't require much in the way of processing power or memory.

In the `provision` section of the playbook, we told Vagrant to provision the all the VMs with Ansible, using the inventory file `inventory`, and the playbook `playbooks/main.yml`. Create these two files in the same folder as your Vagrantfile:

```
deployments-rolling/
  playbooks/
    main.yml
  inventory
  Vagrantfile
```

Inside the `inventory` file, we just need to define a list of all the Node.js API app VMs by IP address:

```
[nodejs-api]
192.168.3.2
192.168.3.3
192.168.3.4
192.168.3.5

[nodejs-api:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

Inside the `main.yml` playbook, we'll call out two separate playbooks—one for the initial provisioning (installing Node.js and making sure the server is configured correctly), and another for deployment (ensuring our Node.js API app is present and running):

```
---
- import_playbook: provision.yml
- import_playbook: deploy.yml
```

Go ahead and create the `provision.yml` and `deploy.yml` playbooks, starting with `provision.yml`:

```
---
- hosts: nodejs-api
  become: yes

  vars:
    nodejs_install_npm_user: root
    npm_config_prefix: "/usr"
    nodejs_npm_global_packages:
      - forever
    firewall_allowed_tcp_ports:
      - "22"
      - "8080"

  pre_tasks:
    - name: Update apt cache if needed.
      apt: update_cache=yes cache_valid_time=3600

  roles:
    - geerlingguy.firewall
    - geerlingguy.nodejs
    - geerlingguy.git
```

This playbook runs on all the servers defined in our inventory file, and runs three roles on the servers: `geerlingguy.firewall` (which installs and configures a firewall, in this case opening ports 22 for SSH and 8080 for our app), `geerlingguy.nodejs` (which installs Node.js, NPM, and the `forever` package (installed globally), which we'll use to run our app as a daemon), and `geerlingguy.git` (which installs `git`, used for app deployment).

Since we're using two roles from Ansible Galaxy, it's best practice to also include those roles in a requirements file so CI tools and others using this playbook can install all the required roles.

Create a `requirements.yml` file in the root folder and add the following:

```
---
- src: geerlingguy.firewall
- src: geerlingguy.nodejs
- src: geerlingguy.git
```

Whenever someone wants to run the playbook, the command `ansible-galaxy install -r requirements.yml` will install all the required roles.

At this point, your project directory should be structured like the following:

```
deployments-rolling/
  playbooks/
    deploy.yml
    main.yml
    provision.yml
  inventory
  requirements.yml
  Vagrantfile
```

Before we run `vagrant up` and see our infrastructure in action, we need to build out the `deploy.yml` playbook, which will ensure our app is present and running correctly on all the servers.

Inside `deploy.yml`, add the following:

```
---
- hosts: nodejs-api
  gather_facts: no
  become: yes

  vars_files:
    - vars.yml
```

Use `become`, and set `gather_facts` to `no` to save a little time during deployments, since our app doesn't require any of the gathered system facts to run.

Since we have a few variables to define, and we'd like to track them separately for easier file revision history, we'll define the variables in a `vars.yml` file in the same directory as the `deploy.yml` playbook:

```
---
app_repository: https://github.com/geerlingguy/demo-nodejs-api.git
app_version: "2.0.0"
app_directory: /opt/demo-nodejs-api
```

Once you've saved the `vars.yml` file, continue building out `deploy.yml`, starting with a task to clone the app's repository (which we just defined in `vars.yml`):

```
  tasks:
    - name: Ensure Node.js API app is present.
      git:
        repo: "{{ app_repository }}"
        version: "{{ app_version }}"
        dest: "{{ app_directory }}"
        accept_hostkey: true
      register: app_updated
      notify: restart forever apps
```

Using variables for the `git` module's `repo` and `version` affords flexibility; app version changes might happen frequently, and it's easier to manage them in a separate `vars.yml` file.

We also want to `notify` a `restart forever apps` handler whenever the codebase is changed. We'll define the `restart forever apps` handler later in the playbook.

```
    - name: Stop all running instances of the app.
      command: "forever stopall"
      when: app_updated.changed

    - name: Ensure Node.js API app dependencies are present.
      npm: "path={{ app_directory }}"
      when: app_updated.changed

    - name: Run Node.js API app tests.
      command: "npm test chdir={{ app_directory }}"
      when: app_updated.changed
```

Once the app is present on the server, we need to use `npm` to install dependencies (using Ansible's `npm` module), then run the app's test suite using `npm test`. To save time, we only stop the application, update dependencies, and run tests if the application has changed (using the `app_updated` variable we registered when checking out the application code).

Running the tests for the app during every deployment ensures the app is present and in a functioning state. Having a thorough unit and integration test suite running on every deployment is prerequisite to a frequent or continuously-integrated project! Running the tests during deployments also enables zero-downtime deployments, as we'll see later.

```
    - name: Get list of all running Node.js apps.
      command: forever list
      register: forever_list
      changed_when: false

    - name: Ensure Node.js API app is started.
      command: "forever start {{ app_directory }}/app.js"
      when: "forever_list.stdout.find('app.js') == -1"
```

Once the app is present and running correctly, we need to make sure it's started. There's a command to get the list of all running apps (using `forever`), then a command to start the app if it's not already running.

```
    - name: Add cron entry to start Node.js API app on reboot.
      cron:
        name: "Start Node.js API app"
        special_time: reboot
        job: "forever start {{ app_directory }}/app.js"
```

The final task adds a cron job to make sure the app is started after the server reboots. Since we're managing the deamonization of our app using `forever` instead of the OS's init system, it's best to make sure the app starts on system boot using a `reboot` cron job.

Remember when we added the line `notify: restart forever apps` to the task that ensured the app was present on the server? It's time to define this handler, which runs the command `forever restartall` (which does exactly what it says):

```
  handlers:
    - name: restart forever apps
      command: "forever restartall"
```

At this point, the Ansible playbooks and Vagrant configuration should be complete. The playbook will clone the `demo-nodejs-api` project, run its tests to make sure everything's working correctly, then start the app using `forever` and make sure it's started whenever the server reboots.

Run the command below to test all the new servers and make sure the app is running correctly:

```
$ for i in {2..5}; \
    do curl -w "\n" "http://192.168.3.$i:8080/hello/john"; \
  done
```

If all the servers are online, you should see the text `"hello john"` repeated four times (once for each server):

```
"hello john"
"hello john"
"hello john"
"hello john"
```

Run `vagrant provision` to run the entire provisioning and deployment process again, or just run `ansible-playbook -i inventory playbooks/deploy.yml` to run the deployment playbook again. In either case, you should see no changes, and Ansible should verify everything is `ok`.

You now have a fleet of Node.js API servers similar to Server Check.in's server checking infrastructure—except it doesn't do much yet! Luckily, the project has seen some new feature development since the initial `2.0.0` version you just deployed. We now need a way to get the new version deployed to and running on all the servers while maintaining 100% uptime for the API as a whole.

Ensuring zero downtime with `serial` and integration tests

Now, after a little extra time in development, we have new features to deploy in a `2.0.1` version. You could run the exact same `ansible-playbook` command as above, adding in `--extra-vars "app_version=2.0.1"`, but best practice is to update the variable in your included variables file, since that change is tracked in version control and used for automated deployments.

Change the `app_version` in `playbooks/vars.yml` to `2.0.1`, and run the deployment playbook again:

```
ansible-playbook -i inventory playbooks/deploy.yml
```

Uh oh—after we deployed the new version, our tests started failing! Since we deployed to all four servers asynchronously, all four application servers are offline, and our boss and customers are going to be very angry.

In this case, rolling back is simple: revert to `2.0.0` and redeploy. Doing this now fixes the problem *this* time, but if part of the application update changed a database schema you could be in a world of hurt!

Ansible has two particular settings to protect you when you deploy to many servers while maintaining your infrastructure's overall integrity during a failed deployment.

Open the deployment playbook (`playbooks/deploy.yml`) and modify the initial settings to match the following:

```
---
- hosts: nodejs-api
  gather_facts: no
  become: yes
  serial: 2
```

Note the addition of `serial: 2`. This tells Ansible to run the entire playbook on two servers at a time. If you update `app_version` to `2.0.1` again, and run the playbook, you should see it run on two of the four servers, and once it hits the test failure, the playbook execution will stop—leaving your other two servers up (and saving you a few hours on a conference bridge explaining the outage).

You could again revert back to `2.0.0`, but in the time you were deploying the failed version, developers finished a new version that got all tests passing again, `2.0.2`. Go ahead and update `app_version` and run the playbook again.

```
PLAY RECAP *********************************************************
192.168.3.2          : ok=8    changed=5    unreachable=0    failed=0
192.168.3.3          : ok=8    changed=5    unreachable=0    failed=0
192.168.3.4          : ok=8    changed=5    unreachable=0    failed=0
192.168.3.5          : ok=8    changed=5    unreachable=0    failed=0
```

Whew! Everything is back online and operational, and all tests are passing with the latest version of the application.

Tests should rarely fail only on production. But there are many times where networking issues or even latency in third party services causes a random failure or two. Whenever you move beyond one server (usually to provide both redundancy and capacity), you will run into these transient issues. It's best to account for them in your automated deployment process by tuning `serial` and similar settings well.

Ansible exposes two different settings for controlling rolling deployment failure scenarios:

1. `serial`: Can be an integer (e.g. `3`) or a percentage (e.g. `30%`). Used to control how many hosts Ansible will manage at once.
2. `max_fail_percentage`: An integer between `1-100`. Used to tell Ansible what percentage of hosts can fail a task before the play will be aborted.

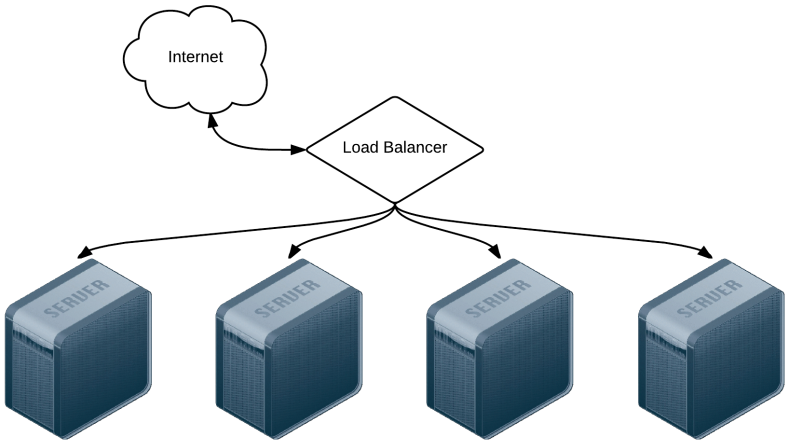
If you have some headroom in your infrastructure, set these values higher. If you have only as much infrastructure running as your application needs, and having more than one or two servers offline would put your infrastructure into a bad state, you should be more conservative with these settings—and maybe provision a little more capacity!

Code for this entire example is available on GitHub in the Ansible for DevOps deployments-rolling example[105].

Deploying to app servers behind a load balancer

In the case of Server Check.in, there are two separate API layers that manage the complexity of ensuring all server checks happen, regardless of whether certain servers are up or down. The 'load balancing' occurs on the application layer instead of as a separate infrastructure layer (this is extremely helpful when dealing with global latency and network reliability variation).

For many applications, especially those with app servers close together (e.g. in the same data center) the infrastructure layer follows a more traditional layout, with a load balancer to handle the API request distribution:



Four servers behind a load balancer.

For a demonstration of zero-downtime deployment with a load balancer, let's build a local infrastructure with one HAProxy load balancer and two Apache webservers.

First, create a new project folder `deployments-balancer`, and within it, create the following Vagrantfile:

[105]https://github.com/geerlingguy/ansible-for-devops/tree/master/deployments-rolling

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  # Base VM OS configuration.
  config.vm.box = "geerlingguy/ubuntu1804"
  config.vm.synced_folder '.', '/vagrant', disabled: true
  config.ssh.insert_key = false

  config.vm.provider :virtualbox do |v|
    v.memory = 256
    v.cpus = 1
    v.linked_clone = true
  end

  # Define four VMs with static private IP addresses.
  boxes = [
    { :name => "bal1", :ip => "192.168.4.2" },
    { :name => "app1", :ip => "192.168.4.3" },
    { :name => "app2", :ip => "192.168.4.4" }
  ]

  # Provision each of the VMs.
  boxes.each do |opts|
    config.vm.define opts[:name] do |config|
      config.vm.hostname = opts[:name]
      config.vm.network :private_network, ip: opts[:ip]

      # Provision all the VMs using Ansible after last VM is up.
      if opts[:name] == "app2"
        config.vm.provision "ansible" do |ansible|
          ansible.playbook = "playbooks/provision.yml"
          ansible.inventory_path = "inventory"
          ansible.limit = "all"
        end
      end
    end
  end

end
```

This Vagrantfile will create three servers running Ubuntu: `bal1` (the balancer), and `app1` and `app2` (the application servers). We referenced an Ansible playbook at `playbooks/provision.yml` (to install the required software on the servers), as well as a custom inventory file at `inventory`. First, create the inventory file (`inventory`, in the same directory as the Vagrantfile), with the appropriate groupings and connection variables:

```
[balancer]
192.168.4.2

[app]
192.168.4.3
192.168.4.4

[deployments:children]
balancer
app

[deployments:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

With this inventory, we can operate on just the `balancer`, just the `app` servers, or all the servers together (in the `deployments` group). Next, create a playbook (at `playbooks/provision.yml`) to provision the servers:

```
---
- hosts: balancer
  become: yes

  vars:
    firewall_allowed_tcp_ports:
      - "22"
      - "80"
    haproxy_backend_servers:
      - name: 192.168.4.3
        address: 192.168.4.3:80
      - name: 192.168.4.4
        address: 192.168.4.4:80

  roles:
    - geerlingguy.firewall
    - geerlingguy.haproxy

- hosts: app
  become: yes

  vars:
    firewall_allowed_tcp_ports:
      - "22"
      - "80"

  roles:
    - geerlingguy.firewall
    - geerlingguy.apache
```

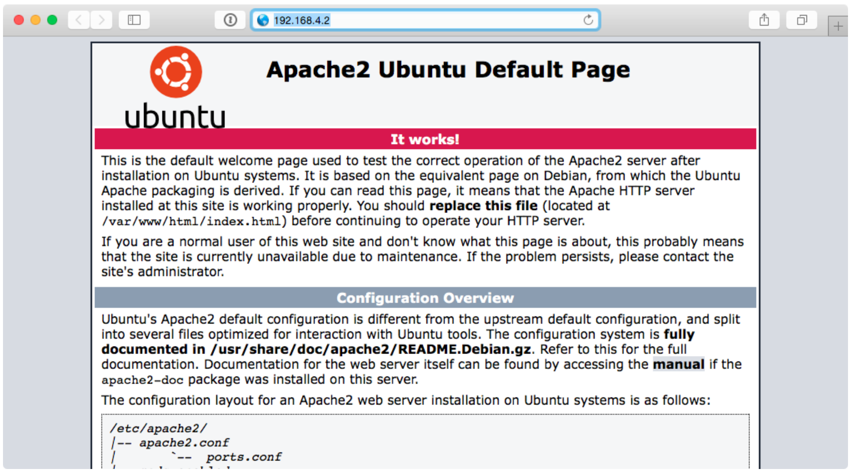
These two plays set up a firewall on both servers, and configure HAProxy on the load balancer, and Apache (with its default configuration) on the app servers. The only required configuration to get this infrastructure working is `haproxy_backend_servers`. We let the `geerlingguy.firewall`, `geerlingguy.haproxy`, and `geerlingguy.apache` roles do all the hard work for us.

Now, to make sure we have all these roles installed, create a requirements file to install the roles from Ansible Galaxy. Create `requirements.yml` in the same directory as the Vagrantfile, with the following contents:

```
---
- src: geerlingguy.firewall
- src: geerlingguy.haproxy
- src: geerlingguy.apache
```

To install the required roles, run `ansible-galaxy install -r requirements.yml`.

At this point, if you want to bring up your local load-balanced infrastructure, run `vagrant up` in the `deployments-balancer` directory, and wait a few minutes. Once everything is up and running, visit `http://192.168.4.2/`, and you should see the default Ubuntu Apache2 landing page:



Apache2 Ubuntu Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Ubuntu systems. It is based on the equivalent page on Debian, from which the Ubuntu Apache packaging is derived. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Configuration Overview

Ubuntu's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Ubuntu tools. The configuration system is **fully documented in /usr/share/doc/apache2/README.Debian.gz**. Refer to this for the full documentation. Documentation for the web server itself can be found by accessing the **manual** if the `apache2-doc` package was installed on this server.

The configuration layout for an Apache2 web server installation on Ubuntu systems is as follows:

```
/etc/apache2/
|-- apache2.conf
|       `--  ports.conf
```

HAProxy is serving requests through the Apache backend servers.

Verify round-robin load balancing is working by running the following command:

```
$ for i in {1..5}; do curl -Is http://192.168.4.2/ | grep Cookie; \
done
Set-Cookie: SERVERID=192.168.4.4; path=/
Set-Cookie: SERVERID=192.168.4.3; path=/
Set-Cookie: SERVERID=192.168.4.4; path=/
Set-Cookie: SERVERID=192.168.4.3; path=/
Set-Cookie: SERVERID=192.168.4.4; path=/
```

You should see the load balancer distributing requests between the two backend app servers.

When you deploy new code to the application servers, you need to guarantee the load balancer always has an app server from which requests can be served, so you want to use `serial` to do the deployment on each server (or groups of servers) in sequence. To make sure the servers are properly removed from HAProxy, then added again post-deploy, use `pre_tasks` and `post_tasks`.

Create another playbook alongside `provision.yml` called `deploy.yml`, with the following contents:

```
---
- hosts: app
  become: yes
  serial: 1

  pre_tasks:
    - name: Disable the backend server in HAProxy.
      haproxy:
        state: disabled
        host: '{{ inventory_hostname }}'
        socket: /var/lib/haproxy/stats
        backend: habackend
      delegate_to: "{{ item }}"
      with_items: "{{ groups.balancer }}"

  tasks:
    - debug: msg="Deployment would be done here."

  post_tasks:
    - name: Wait for backend to come back up.
      wait_for:
        host: '{{ inventory_hostname }}'
        port: 80
        state: started
        timeout: 60

    - name: Enable the backend server in HAProxy.
      haproxy:
        state: enabled
        host: '{{ inventory_hostname }}'
        socket: /var/lib/haproxy/stats
        backend: habackend
      delegate_to: "{{ item }}"
      with_items: "{{ groups.balancer }}"
```

This playbook doesn't do much in terms of actual deployment, but it does illustrate how to do a zero-downtime rolling update over two or more application servers:

1. In `pre_tasks`, the `haproxy` module disables the current `app` server (using the `inventory_hostname` variable) on all the load balancers in the `balancer` group, using `with_items`. The HAProxy task is delegated to each of the `balancer` servers (in our case, only one), since the task affects the load balancer, not the current `app` host.
2. In the `post_tasks`, we first `wait_for` port 80 to be available, and once it is, the `haproxy` module re-enables the current `app` server on all the load balancers.

Run the playbook on the local infrastructure with the following command:

```
$ ansible-playbook -i inventory playbooks/deploy.yml
```

It should only take a few seconds to run, and once it's finished, all the servers should be back in the mix for the load balancer. If you want to quickly confirm the deployment playbook is working as it should, add a task which always fails, immediately following the `debug` task:

```
[...]
tasks:
  - debug: msg="Deployment would be done here."
  - command: /bin/false

post_tasks:
[...]
```

If you run the deployment playbook again, wait for it to fail, then run the `curl` command again, you'll notice all the requests are being directed to the second app server:

```
$ for i in {1..5}; do curl -Is http://192.168.4.2/ | grep Cookie; done
Set-Cookie: SERVERID=192.168.4.4; path=/
Set-Cookie: SERVERID=192.168.4.4; path=/
Set-Cookie: SERVERID=192.168.4.4; path=/
Set-Cookie: SERVERID=192.168.4.4; path=/
Set-Cookie: SERVERID=192.168.4.4; path=/
```

Fix the deployment by removing the `/bin/false` command. Run the playbook one more time to restore the infrastructure to a fully functional state.

This demonstration may seem basic, but the `pre_tasks` and `post_tasks` in the playbook are identical to what many large-scale production infrastructure deployments use!

Code for this entire example is available on GitHub in the Ansible for DevOps deployments-balancer example[106].

Capistrano-style and blue-green deployments

Many developers who deal with Ruby applications are familiar with Capistrano[107], a task automation and application deployment application built with Ruby. Capistrano's basic style of deployment is to create dated release directories, then symlink

[106]https://github.com/geerlingguy/ansible-for-devops/tree/master/deployments-balancer
[107]http://capistranorb.com/

the current release into a stable application directory, along with resources that are continuous among releases (like logs and uploaded files).

Capistrano does a lot more than that basic deployment model, but many people want to replicate a simple application deployment workflow (which also makes rollbacks easy, since you just revert the symlink to the previous release directory!). This is easy to do with Ansible, and rather than walk you through the entire process in this book, I'll point you to a few great resources and an Ansible Galaxy role that coordinates Capistrano-style deployments with ease:

- Rebuilding Capistrano-like deployment with Ansible[108]
- project_deploy role on Ansible Galaxy[109]
- Thoughts on deploying with Ansible[110] (background for the above role)
- Ansible project-deploy[111] (presentation about the above role)

Extending things a little further, many organizations use **blue-green** deployments. The basic concept involves bringing up a parallel production infrastructure, then switching over to it. The cutover may take only a few milliseconds and no active production infrastructure is ever offline during the deployment process.

A few different technologies and concepts, like container-based infrastructure and microservices (which are faster to deploy), and better cloud autoscaling and load balancing options, have made blue-green deployments much easier than in the past.

This book won't go through a detailed example of this style of deployment, as the process is similar to other examples provided, the only difference being an additional task of switching a load balancer from the old to the new infrastructure once it's up and running. Ansible's blog has an excellent overview of AWS-based blue-green deployments: Immutable Systems and Ansible[112], and there are built-in modules to manage almost any type of load balancer you could use, including F5's BIG-IP[113], HAProxy[114], Citrix NetScaler[115], and Amazon ELB[116].

[108]http://blog.versioneye.com/2014/09/24/rebuilding-capistrano-like-deployment-with-ansible/
[109]https://github.com/f500/ansible-project_deploy
[110]http://www.future500.nl/articles/2014/07/thoughts-on-deploying-with-ansible/
[111]http://www.slideshare.net/ramondelafuente/ansible-projectdeploy
[112]http://www.ansible.com/blog/immutable-systems
[113]http://docs.ansible.com/list_of_network_modules.html#f5
[114]http://docs.ansible.com/haproxy_module.html
[115]http://docs.ansible.com/netscaler_module.html
[116]http://docs.ansible.com/ec2_elb_module.html

Additional Deployment Features

There are a few other Ansible modules and options which are helpful in the context of deployments:

`run_once`[117] and `delegate_to` are extremely helpful in scenarios like updating a database schema or clearing an application's cache, where you need a particular task to only run one time, on a particular server:

```
- command: /opt/app/upgrade-database-schema
  run_once: true
  delegate_to: app1.example.com
```

Using `run_once` with `delegate_to` is similar to the pattern of using `when: inventory_hostname == groups.groupname[0]`, but is a little more precise in describing what you're trying to achieve—running a command once on a specific host.

Another important aspect of a successful deployment is communication. If you're running playbooks as part of a CI/CD process, or in some other automated fashion, use one of the many built-in Ansible notification modules to share the deployment's progress via chat, email, or even text-to-speech on your Mac with the `osx_say` module! Ansible includes easy-to-use notification modules for:

- Campfire
- HipChat
- IRC
- Jabber
- Email
- Slack
- Twilio
- Amazon SNS
- etc.

Many playbooks include notifications in both the `pre_tasks` and `post_tasks` sections, notifying admins in a chat channel when a deployment begins or ends. For example:

[117] http://docs.ansible.com/playbooks_delegation.html#run-once

```
post_tasks:
  - name: Tell everyone on IRC the deployment is complete.
    irc:
      channel: my-org
      server: irc.example.com
      msg: "Deployment complete!"
    delegate_to: 127.0.0.1
```

For a great primer on Ansible notifications, see Ansible Inc's blog post: Listen to your Servers Talk[118].

Summary

Automating deployments with Ansible enables your development team to have their code on production servers more reliably and quickly, and it enables your operations team to spend less time on repetitive tasks, and more time improving your infrastructure.

This chapter outlined only a few of the most popular deployment techniques, but Ansible is flexible enough to handle almost any situation out of the box.

```
 _______________________________________
/ One machine can do the work of fifty  \
| ordinary men. No machine can do the   |
| work of one extraordinary man.        |
\ (Elbert Hubbard)                      /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[118] http://www.ansible.com/blog/listen-to-your-servers-talk