Joining the AWS realm

We finished the previous chapter with a fully functional application that we are ready to deploy. This chapter explains how to use Ansible to deploy the sample application we just created into Amazon EC2. The steps describe both how to deploy the application and how to configure the AWS components required to run the application, like RDS or DynamoDB, via AWS CloudFormation.

If we recover the sketched project plan we defined at the beginning of the last chapter, we had some steps to follow:

1. Setup a development environment, including Amazon AWS services you may use
2. Develop the code
3. Setup a staging area (development) in Amazon AWS
4. Setup a test environment in Amazon AWS
5. Setup a production environment in Amazon AWS
6. Launch the application
7. Scale it as the demand grows
8. Profit!

This chapter covers steps 3, 6 and 7. As we mentioned before, steps 3, 4 and 5 follow almost identical steps, which means that this chapter can be used to deploy additional environments as defined by points 4 and 5.

We also cover part of step 1. As it was said at the beginning of last chapter, we moved all the AWS related steps to this chapter. This includes the steps to use CloudFormation to configure DynamoDB, which you need to run the sample application in a development environment.

In the following sections we talk about Ansible for deployments, we explain the scripts in this example and finally we show how to deploy the application using the scripts.

Remember that the scripts are available in Github. They can be downloaded and modified as required to help you deploy the sample application in EC2.

Using Ansible for deployments

Ansible is a tool for DevOps that allows you to remotely execute commands in your servers. It provides a modular system in which you can define individual tasks to run according to the server role. Ansible complements the scripts with a thorough API that helps with common operations and a set of plugins for popular components (Amazon AWS, PostgreSQL, etc.).

In this article we use Ansible v1.3. This version (currently under development) expands the library of services related to Amazon AWS. The new components facilitate new operations like creating an AMI from an existing EC2 instance, something we need for our project.

The aim of the Ansible scripts is to automate the process of creating a custom AMI and using that AMI in CloudFormation to automate the deployment of the application. The scripts can also be used to release upgrades of the given application.

Configuring Ansible

Ansible has minimal requirements; the only piece of software necessary on your machine to launch Ansible is Python 2.6 (or greater) and a few Python modules. This facilitates its deployment in any development or system administration computer. The connection with the servers is established via

SSH, a common protocol included by default on all Linux machines.

Although we don't explain how to install Ansible in this document, as it depends on your system and the official website has excellent documentation, we want to bring your attention to a couple of common issues. The first one is that Ansible 1.3 requires the latest versions of some Python modules, namely *boto*, *PyYAML, Jinja2* and *paramiko.* Your local environment may need to be upgraded or you may get execution errors when running the Ansible Playbooks.

Another possible issue is the management of AWS access keys. Ansible can read the AWS access and secret keys from system environment variables, but in some cases one of the Python modules used by Ansible, *boto*, may not find them. This means that you need to define a few environment variables in your system before running the scripts:

```
1 # Exports that may be needed in your system
2 export EC2_ACCESS_KEY=<key>
3 export EC2_SECRET_KEY=<secret_key>
4 export AWS_ACCESS_KEY_ID=<key>
5 export AWS_SECRET__ACCESS_KEY=<secret_key>
6 export BOTO_PATH=~/.boto/boto.cfg
```

Lines 2 to 5 define necessary environment variables that contain the AWS access and secret key. Line 6 defines a variable that points to *boto* configuration. The file follows the format:

```
# BOTO config
[Credentials]
aws_access_key_id = <key>
aws_secret_access_key = <secret_key>
```

and it will be used by *boto* to authenticate Ansible requests against services like S3.

Structure of Ansible Folder

Ansible uses a series of scripts to define the instructions to run on a given server. The scripts are called Playbooks and they follow the YAML format. The Github repository contains the Ansible Playbooks we use to deploy the sample application. If you explore the folder you may notice that there is a certain structure in it. The folders and files are organized following Best Practices for Ansible.

Using best practices is not just a recommendation but almost a mandatory requirement. Many Ansible commands (on version 1.2 and onwards) assume some files will be located in the locations recommended by the best practices by default. Following these recommendations saves you time when trying to use these commands, and keeps the Playbooks nicely organized.

If you open the folder with the scripts, you may see the following contents:

- **groups_vars** (folder): contains configuration variables for the Playbooks
- **roles** (folder): contains the role Playbooks, modules with reusable functionality
- **webserver_setup.yml:** defines the roles to execute when creating our custom AMI
- **cloudformation_setup.yml:** defines the roles to execute when launching CloudFormation
- **local.hosts:** a dummy hosts file used to launch our Ansible scripts against AWS.
- **Readme.md:** some notes on the contents of the folder

In the root folder there are two main Playbooks available. One of them is used to create a custom AMI, while the other launches a CloudFormation group. These Playbooks contain a list of *roles* to run, in order of execution. The *roles* are the ones doing the real work, while the main Playbooks just list which ones we trigger. This division makes it very easy to change the steps being executed in a Playbook, by adding or removing roles.

The *group_vars* folder has a file named *all*. This file contains a list of pairs (key: value) which define variables and values. These variables are then used in the Playbooks, replacing each key by its corresponding value. This allows us, for example, to define the version of Play Framework we want to install and to change it later on by editing only one location.

The *roles* folder contains the different '*roles*'. A role is a set of Playbooks, templates and other files that do a specific task in a server. For example, the '*create*' role has a list of commands to create an EC2 instance. This division on roles facilitates reusing tasks across Playbooks, and we can enable or disable specific tasks for a given Playbook at any moment with minimal effort.

Ansible Scripts

A full description of an Ansible script is outside the scope of this document, please refer to the documentation for this. This section provides a quick overview of the basics of a script.

An Ansible script is a YAML file, which is plain text and human readable, that contains a set of commands to execute in the target servers. An example:

```
 1 # Secures an Ubuntu instance.
 2 #
 3 ---
 4 - name: Update APT package cache
 5   action: apt update_cache=yes
 6 
 7 - name: Run apt-get upgrade
 8   action: apt upgrade=yes
 9 
10 - name: Install fail2ban
11   action: apt pkg=fail2ban state=installed
```

In the sample above line 3 indicates the start of the script. As you can see, we have three actions defined by a *name* and the *action* itself. The *name* is displayed during the execution of the Playbook, allowing us to know which step is being run. The *action* corresponds to either an Ansible command, using the modules and API provided by the tool, or a raw command executed via *SSH*.

The steps are run in order. If none of them fail, the script finishes with a status of success. If a step fails and we have not opted to ignore errors in that particular step, then the script will abort and we will be notified of the error. Having errors in one server may not mean that the execution will stop in other servers.

When writing Playbooks, be aware that Ansible is not atomic. That is, if the execution of the Playbook fails in a server Ansible doesn't rollback any step that has been executed in that server. This means that it is recommended to add steps that can be run multiple times or with proper safeguards. The script may fail in the middle of execution and you may need to execute it again on the same server, which results in some steps being run twice.

Explaining the Scripts

This section gives some details on the Ansible Playbooks used to create the custom AMI and to execute the CloudFormation template. We don't explain the code itself, as the scripts are documented in the source files, but we comment on the purpose of each script and how it works.

We have broken the process in two parts (two Playbooks) for practical reasons. One Playbook is used to create a custom AMI that contains the web application. The second is used to configure any required AWS resources and to launch the application.

By providing a Playbook to create a custom AMI based on an initial Ubuntu Linux AMI and the source of the application, we facilitate reusing the script. We can execute that same Playbook to create multiple AMI in multiple regions just by changing some configuration. This allows us to have multiple AMI ready for deployment, each one with specific settings, with minimal effort.

Deploying the system live is another task, bigger in scope and more costly economically due to all the resources needed in the cloud (databases, servers, etc. which have an hourly cost). And it's done sparingly, only to start specific projects. Thus, it makes sense to separate this step and provide a single Playbook that manages it. Again, this will facilitate using the script multiple times as we can specify the target AMI or the CloudFormation components for each execution without much effort.

With this, we can launch our application in AWS just by typing two commands.

Creating a custom AMI

The *webserver_setup* Playbook is used to create a custom AMI, based in Ubuntu 13.04, that is secured against external attacks and contains the web application ready to be deployed.

The Playbook executes 4 roles for this purpose:

- **create:** this role creates a new EC2 *m1.small* instance and connects to it via SSH so the other roles can update it as required
- **secure:** this role secures the instance by installing security applications and enabling some access restrictions
- **deploy:** this role deploys the sample application to the instance. It uses configuration options to decide which version of Play Framework to install and where to find the source code, compiles it and deploys it in the server
- **saveAMI:** this role saves the EC2 instance we have configured into a custom AMI, so we can create multiple instances with the exact same configuration

You should not need to modify the scripts, all the relevant configuration can be set via the Ansible *all* file, as explained in the next section.

CloudFormation

The *cloudformation_setup* Playbook is used to execute a CloudFormation template in AWS and deploy a full application stack. The advantage of using CloudFormation is automating most of the process.

Usually we would need to define a DynamoDB database, a new MySQL RDS instance, AutoScaling, Load Balancing, etc. Lots of steps to be run manually for each deployment, very time-consuming. Using CloudFormation we simply define all this in a template which we execute so that AWS does all the required work for us. It simplifies the process and makes it easier to apply changes to it.

The Playbook has two requirements. The first is a custom AMI with the web application to deploy. In this case that is the custom AMI created by the Playbook described in the previous section, whose *id* we can add as a configuration entry.

The second requirement is the CloudFormation template itself. That is a JSON file that describes all the AWS components required, and that we can customize via some parameters. Ansible makes this part easy by including a task that allows you to indicate a template to run and the parameters to provide to the template.

Deploying the Application

This section explains how to run the scripts to create our custom AMI containing the project created in the previous article.

Configuring the Scripts

The scripts are mostly self contained. This means that you should not need to modify the Ansible Playbooks themselves, but there are some configuration settings to be adjusted for your own deployment.

The first step is to obtain an Amazon KeyPair. This is an important step as without a KeyPair you can't connect remotely to AWS. The official AWS documentation explains how to obtain a new KeyPair or download an existing one.

Once you have downloaded the KeyPair, you have to modify its attributes to be read only, and then add it to SSH via *ssh-add*:

```
$ chmod 600 sample-play-aws.pem
$ ssh-add sample-play-aws.pem
```

The next step is to ensure that all the Ansible configuration is as expected. We have 3 main files to check:

- **group_vars/all:** this file contains values used across all the Playbooks. Values include the AWS region to use for deployment, the name of the KeyPair for AWS operations or the version of Play Framework to download. Ensure the values are correct before proceeding.
- **roles/deploy/files/start:** this is the *start* script used to launch the web application. Add the proper values to connect to the RDS db and the AWS credentials via *-D* JVM properties. If not updated properly, the application won't start.
- **roles/cloudformation/files/PlayFrameworkApp.json:** this file contains a CloudFormation template. You can edit it to add or remove CloudFormation elements from it.

As a clarification, there is one variable in *group_vars/all* that can't be updated properly until we run the first step of the process. That variable is *cf_ami* and points to the AMI created by the first Ansible Playbook. The value is only relevant when using the CloudFormation Playbook.

Running the Scripts

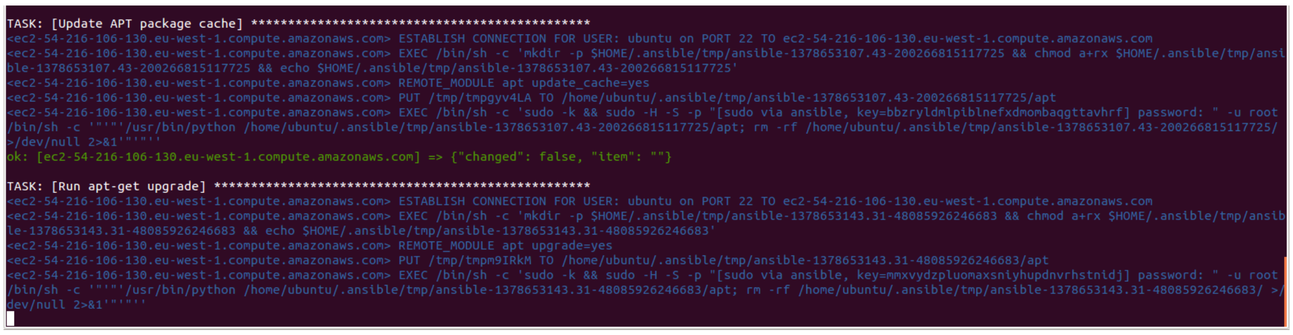
To run the scripts you need to have Ansible installed. If you have not installed it yet, please follow the instructions corresponding to your development environment. Ansible 1.3 is required to run the scripts in this example. We assume that the configuration details, as described above, are correct.

Is at this point when one sees the advantages of using Ansible versus other systems. We have to execute only two scripts, corresponding to the two steps: create a custom AMI and use CloudFormation to deploy the application. The scripts automate all the process, freeing you from a time consuming work so you can focus on other tasks.

To run the first script, go to the folder where you have deployed your Ansible files and execute the following command:

$ ansible-playbook -i local.hosts --private-key=<path_to_pem_file> webserver_setup.yml

This will launch the Playbooks to create a custom AMI that contains our application, and it will take several minutes. An example of the output is:



```
TASK: [Update APT package cache] *************************************************
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> ESTABLISH CONNECTION FOR USER: ubuntu on PORT 22 TO ec2-54-216-106-130.eu-west-1.compute.amazonaws.com
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> EXEC /bin/sh -c 'mkdir -p $HOME/.ansible/tmp/ansible-1378653107.43-200266815117725 && chmod a+rx $HOME/.ansible/tmp/ansi
ble-1378653107.43-200266815117725 && echo $HOME/.ansible/tmp/ansible-1378653107.43-200266815117725'
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> REMOTE_MODULE apt update_cache=yes
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> PUT /tmp/tmpgyv4LA TO /home/ubuntu/.ansible/tmp/ansible-1378653107.43-200266815117725/apt
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> EXEC /bin/sh -c 'sudo -k && sudo -H -S -p "[sudo via ansible, key=bbzryldmlpiblnefxdmombaqgttavhrf] password: " -u root
/bin/sh -c '"'"'/usr/bin/python /home/ubuntu/.ansible/tmp/ansible-1378653107.43-200266815117725/apt; rm -rf /home/ubuntu/.ansible/tmp/ansible-1378653107.43-200266815117725/
>/dev/null 2>&1'"'"''
ok: [ec2-54-216-106-130.eu-west-1.compute.amazonaws.com] => {"changed": false, "item": ""}

TASK: [Run apt-get upgrade] *************************************************
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> ESTABLISH CONNECTION FOR USER: ubuntu on PORT 22 TO ec2-54-216-106-130.eu-west-1.compute.amazonaws.com
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> EXEC /bin/sh -c 'mkdir -p $HOME/.ansible/tmp/ansible-1378653143.31-48085926246683 && chmod a+rx $HOME/.ansible/tmp/ansib
le-1378653143.31-48085926246683 && echo $HOME/.ansible/tmp/ansible-1378653143.31-48085926246683'
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> REMOTE_MODULE apt upgrade=yes
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> PUT /tmp/tmpm9IRkM TO /home/ubuntu/.ansible/tmp/ansible-1378653143.31-48085926246683/apt
<ec2-54-216-106-130.eu-west-1.compute.amazonaws.com> EXEC /bin/sh -c 'sudo -k && sudo -H -S -p "[sudo via ansible, key=mmxvydzpluomaxsniyhupdnvrhstnidj] password: " -u root
/bin/sh -c '"'"'/usr/bin/python /home/ubuntu/.ansible/tmp/ansible-1378653143.31-48085926246683/apt; rm -rf /home/ubuntu/.ansible/tmp/ansible-1378653143.31-48085926246683/ >/
dev/null 2>&1'"'"''
```

Once the script has run, log into your AWS Console to retrieve the *id* of the custom AMI and add it to the configuration file as the value for the *cf_ami* variable. This tells the CloudFormatio Playbook which AMI to use for the setup.

To deploy the application using CloudFormation, execute:

$ ansible-playbook -i local.hosts --private-key=<path_to_pem_file> cloudformation_setup.yml

The script launches a CloudFormation stack, which may take several minutes.

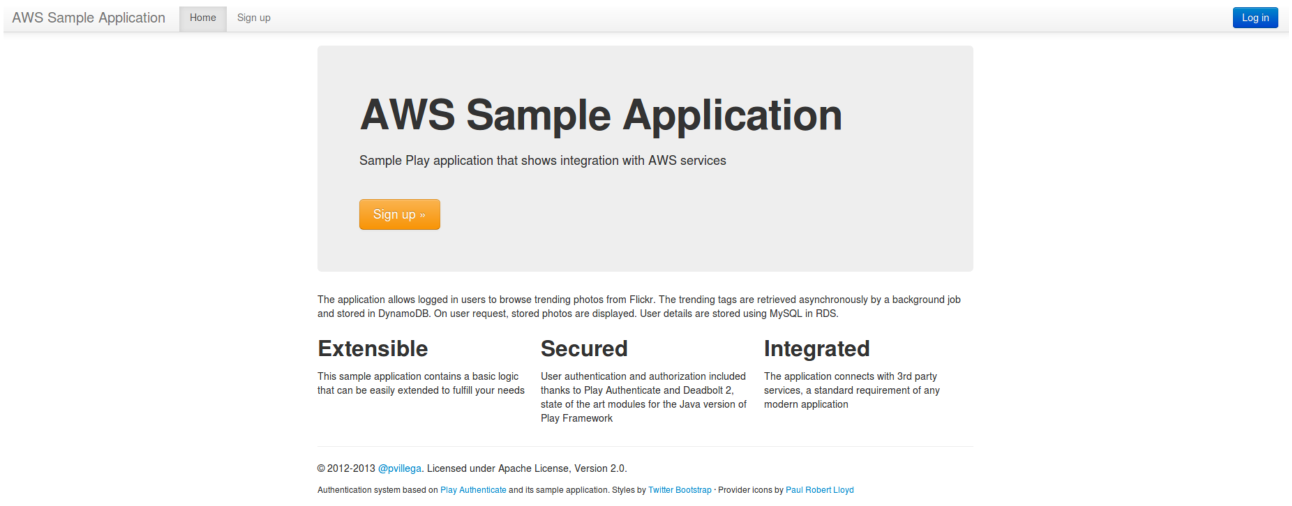
After running

After executing the scripts, you can go to your AWS Console to verify that all the components have been created. In particular you may want to visit the CloudFormation dashboard, where you can see a list of the components Ansible launched.

Having created an Elastic Load Balancer as part of the process, you can obtain its public Elastic IP. This allows you to map the balancer to a custom domain via Route 53 or your own DNS management system.

Testing the deployment

At this stage we have deployed the application in EC2 and enabled scaling based on demand. If we access the application via the custom domain we associated to our Elastic Load Balancer, we see the main page of the application:


AWS Sample Application Home Sign up Log in

AWS Sample Application

Sample Play application that shows integration with AWS services

Sign up »

The application allows logged in users to browse trending photos from Flickr. The trending tags are retrieved asynchronously by a background job and stored in DynamoDB. On user request, stored photos are displayed. User details are stored using MySQL in RDS.

Extensible

This sample application contains a basic logic that can be easily extended to fulfill your needs

Secured

User authentication and authorization included thanks to Play Authenticate and Deadbolt 2, state of the art modules for the Java version of Play Framework

Integrated

The application connects with 3rd party services, a standard requirement of any modern application

© 2012-2013 @pvillega. Licensed under Apache License, Version 2.0.

Authentication system based on Play Authenticate and its sample application. Styles by Twitter Bootstrap · Provider icons by Paul Robert Lloyd

To test the scaling of the application I recommend using Siege. Usage of Siege is outside the scope of this document, but you can find an example in here.

This concludes the second chapter of the article. At this point we have created a Play Framework application that interacts with Amazon AWS and we have deployed this application in Amazon EC2 using Ansible. The next chapter explains how to deploy the same application in Elastic BeanStalk.