Exploring AWS DevOps

Experimentation in
Elasticbeanstalk
and
OpsWorks

by Alan T. Landucci-Ruiz

Exploring AWS DevOps

Experimentation in ElasticBeanstalk and OpsWorks

Alan T. Landucci-Ruiz

This book is for sale at http://leanpub.com/awsdevopsexplore

This version was published on 2017-01-04

Contents

1. Identity & Access Management (IAM)

Most of the access control that AWS uses is controlled by Identity & Access Management. When you first start using AWS, it's a good idea to start at the IAM panel and get familiar with everything. You should also create an administrative user so that you're not constantly using the root account. The "Security Status" section of the Dashboard will help walk you through all the necessary steps to ensure your account is properly secured. For DevOps and DevOps support, users will need to be created for the DevOps team and the support staff (AWS Administrators).

Best practices recommends creating a group first, with all of the rights that you think you would need. I started by creating an Administrators group, which has an attached policy named "AdministratorAccess"; I can adjust this later or create new ones, if needed. If you're unsure about the groups you'll need, think about the modules the users might need, then click on the "Access Advisor" tab. For example, I would want to give a DBA access to RDS, but wouldn't want them having access to EC2. A Build Engineer might only need access to OpsWorks or CloudFormation, but not the EC2 services.

Finally, create the User accounts (be sure to create access keys and distribute appropriately) and specify a group for each. Once you've created the accounts, make a note of the IAM users sign-in link at the top of the Dashboard (this can also be customized to your own preference). Copy the link, then sign out. Paste the link into the location bar of your browser, then sign in with your new credentials. If your access is correct, you should now be able to go back to the IAM page and add more users. You can generate passwords for the accounts if they will need to log into the AWS console, but these will otherwise not be needed, since the majority of AWS uses access keys.

2. Exploring Elastic Beanstalk

Introduction

There are many advantages to using Elasticbeanstalk. As a DevOps for simple applications in PHP, Java, Node.js, it has scaling abilities that developers shouldn't need to worry about. It reduces the costs associated with time lost in operating costs. In addition to the advantages, there are many advanced features that come with it. In this exploration, I intend to look at some of these features and how they can be applied to an existing development environment.

Before we begin, let's start with a simple explanation of Elastic Beanstalk. Let's assume each of our applications is considered its own container. We have an application called "Web Notifier." Within the "Web Notifier" container, we can create different environments: dev (developer), prod (production), uat (user applied testing), qa (quality assurance). We can actually have as many environments as we need, but every environment creates a new web space, or EC2 container[1]. Each environment can house several configurations, and each configuration can be applied to each environment. In fact, any setting for the container itself can be applied to any environment, and many are inherited.

Our experimentation and exploration need goals to be defined. The requirements I'm looking for in my environment are as below:

1. A Development Stack that is consistent across all environments.
2. Limits change in our development workflow, or supports our current development workflow:
    1. Git availability
    2. Support existing large site (>13G).
    3. Local accounts on the system.
3. Easily promote publication between environments.
4. PHP Customization in the .ini file.

Items 2c and 3 are ideal if they can be done on-the-fly, but are acceptable "during build". Also, we aren't hard-locked into keeping our current development workflow, as long as a migration to a new workflow is easy. This is because Elasticbeanstalk advertises being able to easily publish to environments.

[1]EC2 containers are similar to ESX's Virtual Machines.

Experimentation

Experiment 1: Exploration of the CLI

Experimentation

During the course of this exploration, we're going to use the Elastic Beanstalk command-line interface (CLI). The installation instructions for the CLI can be found at https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb3-init.html. For simplicity, since our exploration will use a RedHat Enterprise version of Linux, I've included the steps for installing the CLI on RedHat. If you're using a different distribution of Linux, please follow the aforementioned link.

```
$ sudo yum -y install python
$ curl -O https://bootstrap.pypa.io/get-pip.py
$ sudo python ./get-pip.py && rm ./get-pip.py
$ sudo pip install awsebcli
```

We should also install the AWS CLI, which gives us command-line interface to all of AWS tools:

```
pip install awscli
```

The AWS CLI will allow us to do advanced CLI commands, such as destroying full applications. After installing the AWS CLI, we will need to create the credentials to allow our computer's CLI access to make changes to the cloud.

From your AWS console, click on Identity and Access Management. Rather than using accounts or usernames, AWS uses identities and secret access keys for allowing and disallowing permissions to specific areas within AWS, such as S3 or EC2 containers. AWS designates these as Identities because, sometimes, these will be assigned to applications themselves, rather than people, but for all practical purposes, these AWS Identities are also called Users. These identities can also be allowed specific permissions, such as read-only, to each area.[^chap2_2] Thus, with some advanced IAM JSON coding, you can give your developers permissions to write to the development area of Elasticbeanstalk, but not give them the freedom to create any environment they want. To ease our frustrations with managing these identities, AWS allows us to create groups. We will create our first group, named "EBAdmins", and give it the AWSElasticBeanstalkFullAccess. Now that we have our new group, we can create a new Identity by clicking "Users" (remember, in AWS, "Identity" and "User" are interchangeable).

Note that in the User creation pane, we can add up to 5 users at once, and the User Name can be any descriptive name that we want, as long as it is less than 65 alphanumeric characters (and can include any of +=,.@-_). I'll call mine "Ashley.J.Williams."

The user is created and it gives me the User Security Credentials page. It's a good idea to download the credentials of the user(s) that we just created, because if we lose them, we will have to re-create them. We will not need to download new credentials if we change any of the user's permissions. We can also click on "Show User Security Credentials" and copy them down, but we'll download them, which creates a "credentials.csv" file that we can later parse through using a script to create other files that we need. If you created these for other people, then split them out into their own file, encrypt the file, then send the encrypted credentials to the intended user. A screenshot walkthrough can be found in Appendix A.

To use these credentials in the Elasticbeanstalk CLI, we need to create a file that houses our credentials. The filename is called "aws_credential_file" and is placed in the .elasticbeanstalk directory of our home directory. In this example, we've put the credentials.csv in our home directory.

```
$ cd ~
$ mkdir .elasticbeanstalk
$ awk -F, \
> '/Ashley/{NAME=$1;KEYID=$2;SECRET=$3;print "AWSAccessKeyId="KEYID"\nAWSSecretKey="SECRET}' \
> credentials.csv > .elasticbeanstalk/aws_credential_file
$ chmod 600 .elasticbeanstalk/aws_credential_file
$ rm credentials.csv
```

The chmod line sets the file so that only the current user can read it. If you've installed the AWS CLI, you will also need to create the AWS credentials.

```
$ aws configure
AWS Access Key ID [None]: AKIAJE2DXPZPFRXQFQAA
AWS Secret Access Key [None]: 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
Default region name [None]: us-west-2
Default output format [None]:
```

Let's start with a clean (simple) application. In our application directory, we will want to remove any existing .git, .elasticbeanstalk, and .ebextensions subdirectories. If you've never worked with Elasticbeanstalk, you won't have these last two subdirectories.

```
$ cd ~
$ cd WebNotifier
WebNotifier $ rm -r .git
WebNotifier $ rm -r .elasticbeanstalk
WebNotifier $ rm -r .ebextensions
```

Now that we have our directory ready, we can start configuring the Elasticbeanstalk. One note before we begin: login to the AWS console and take a look at the Elasticbeanstalk configuration there. We want to make sure there isn't already a project with the same name; if there is, we'll want to pick a different name for our application. When we initialize our Elasticbeanstalk from the CLI, it detects any other applications that we have created (including ones we've already destroyed) and will ask us if we want to use those or create a new one. It will also ask us for a default region. This is generally the region that is closest to you. For mine, I would select us-west-2 (Oregon). Information on the different "eb" subcommands can be found with the "–help" switch and at http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-getting-started.html.

```
WebNotifier $ eb init

Select a default region
1) us-east-1 : US East (N. Virginia)
2) us-west-1 : US West (N. California)
3) us-west-2 : US West (Oregon)
4) eu-west-1 : EU (Ireland)
5) eu-central-1 : EU (Frankfurt)
6) ap-southeast-1 : Asia Pacific (Singapore)
7) ap-southeast-2 : Asia Pacific (Sydney)
8) ap-northeast-1 : Asia Pacific (Tokyo)
9) ap-northeast-2 : Asia Pacific (Seoul)
10) sa-east-1 : South America (Sao Paulo)
11) cn-north-1 : China (Beijing)
(default is 3): 3


Select an application to use
1) ad1app
2) awsmigrate
3) [ Create new Application ]
(default is 3): 3


Enter Application Name
```

```
(default is "WebNotifier"): <enter>
Application WebNotifier has been created.


It appears you are using PHP. Is this correct?
(y/n): y


Select a platform version.
1) PHP 5.4
2) PHP 5.5
3) PHP 5.6
4) PHP 5.3
(default is 1): 3
Do you want to set up SSH for your instances?
(y/n): y


Select a keypair.
1) awskeypair
2) [ Create new KeyPair ]
(default is 10): 1
```

Elasticbeanstalk detected that the application, from the files in my current directory, was a PHP application. With the "eb init" command, we can control other things like the operating system version, but this simple init subcommand gives us the latest Amazon Linux distribution. I've selected PHP 5.6 because I know my app supports it, but you may want to check the requirements for your application. After running this command, it's created a .gitignore file and an .elasticbeanstalk directory with a file named "config.yml". This file can also be created before you run "eb init", and it will take the configuration syntax from the "config.yml" file to create the application. This file is also where you can specify your OS version. Our auto-generated file is very small with very few specifications.

```
WebNotifier $ cat .elasticbeanstalk/config.yml
branch-defaults:
  default:
    environment: null
    group_suffix: null
global:
  application_name: WebNotifier
  default_ec2_keyname: awskeypair
  default_platform: PHP 5.6
  default_region: us-west-2
  profile: eb-cli
```

```
    sc: null
```

The config files are written in YAML syntax, which uses the spaces as delineators within each sub-group. The advantage over JSON is that we don't have to worry about whether or not we've closed a brace, bracket, or quotation mark (okay, we do have to worry about closing quotation marks). Before we create the environment, we're going to add a couple of users to our configuration. This now requires us to use the .ebextensions directory. The documentation to the .ebextensions and everything that you can do can be found here: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customize-containers-ec2.html.

3. Exploring AWS OpsWorks

Introduction

Now that we've determined that AWS Elasticbeanstalk won't work for our DevOps environment, we need to take a step back and reconsider our architecture. But first, let's go over the requirements I'm looking for in my environment:

1. A Development Stack that is consistent across all environments.
2. Limits change in our development workflow, or supports our current development workflow:
    1. Git availability
    2. Support existing large site (>13G).
    3. Local accounts on the system.
3. Easily promote publication between environments.
4. PHP Customization in the .ini file.

As in our Elasticbeanstalk system, items 2c and 3 are ideal if they can be done on-the-fly, but are acceptable in a "during build" capacity.

Our architecture actually requires a centralized location for the PHP files, since we'll be load balancing. In a unix environment, this typically uses NFS to share out its drives to multiple systems. These systems run the same software, where the load balancer manages incoming traffic to balance it between each system, so that one system doesn't get overloaded with all the requests. One of the drawbacks from using OpsWorks over Elasticbeanstalk is that OpsWorks doesn't do autoscaling. If the systems become overloaded, we will have to manually create a new instance.

OpsWorks uses Chef recipes for building the systems. What is Chef, then? It's a deployment tool, similar to Puppet and Ansible, but its platform is different. Chef allows a builder to deploy software across multiple systems using standard templates, also known as recipes. For more information about Chef, go to https://www.chef.io/. Pluralsight has a few good classes on Chef that you might want to check out if you'd like to learn more.

AWS Opsworks works by either implementing its own pre-built recipes, or by allowing the user to provide one's own Chef recipes. To use Chef recipes on OpsWorks, the recipes need to be available to AWS through either an S3 bucket or a Git repository. For our purposes, we'll use BitBucket, but we could use GitHub or a local GitLab server.

In order for AWS Opsworks to take advantage of your recipes, these need to be stored in an accessible location, such as a git repository (GitHub, GitLab, BitBucket), a subversion repository, an HTTP Archive, or an S3 Archive, with each cookbook being at the root level. Within each cookbook are recipes specific for that book. For example, I have a git repository that has a php cookbook with the following contents:

- php/
  - attributes/
  - CHANGELOG.md
  - CONTRIBUTING.md
  - files/
  - libraries/
  - MAINTAINERS.md
  - metadata.json
  - metadata.rb
  - providers/
  - README.md
  - recipes/
  - resources/
  - templates/

This is just the first level; there are other files further below in the directory tree, but I won't bore you with those contents. If you're interested in them, they can be found in the PHP cookbook in the Chef Supermarket (https://supermarket.chef.io/cookbooks/php). If you browse the Chef Supermarket, you'll find that a lot of the work has already been done for you, as far as standard recipes go. When browsing the Chef Supermarket, always verify that the platform for the recipe you're browsing has been tested on Amazon AWS; advanced options allow you to filter by platform.

For our experiments, we'll want to set up our BitBucket account so that OpsWorks can access our Chef recipes. If you don't want your repository to be public, you'll need to set up deployment keys. In your bitbucket account, create the SSH keys and add them as deployment keys (Settings->Deployment Keys). If you're using Github or Gitlab instead, follow the instructions for deployment keys in its documentation. The application of the keys are the same: public key goes to the Git server, while the Private key will gets imported into AWS (more on this later).

```
$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/alanducci/.ssh/id_rsa): /home/alanducci/bitbucket
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/alanducci/bitbucket.
Your public key has been saved in /home/alanducci/bitbucket.pub.
The key fingerprint is:
SHA256:dFV2pBe07vQEapUCy6C9kYzzS5vRK2mTBFKGxpxwQzs
```

```
The key's randomart image is:
+---[RSA 2048]----+
|  .=+oo . . ..++o|
|   .*= = + + . +o|
|   .E = * + . =..|
|     o = =   +.o |
|        S . o  o.|
|       o B o  o..|
|        X .    ..|
|       . o       |
|                 |
+----[SHA256]-----+


$ cat /home/alanducci/bitbucket
-----BEGIN RSA PRIVATE KEY-----
MII[REDACTED]gY=
-----END RSA PRIVATE KEY-----


$ cat bitbucket.pub
ssh-rsa AAAA[REDACTED]R1
$
```

Copy the public key into the bitbucket account and give it a name. The private key will be needed later to copy into your stack settings.

If you're already familiar with Chef, then most of this will come easy. If you're not familiar with Chef, and have some programming skills, this will probably still come easy. AWS Opsworks removes the knife command from the recipe. This isn't to say that the knife command can't be used to verify your recipes, but rather, the deployments are pushed by the AWS Opsworks console or command-line.

Experimentation

Experiment 1: NFS Server

Experimentation

We'll start with the simplest experiment, an NFS server. Within the Chef Supermarket, there is an NFS cookbook, but we're going to build out our own recipe. One reason we do this is because many of the Chef Supermarket cookbooks rely on environments or roles, which AWS Opsworks does not entirely support. Rather, AWS parallels Chef roles with AWS OpsWorks Layers. AWS OpsWorks also provides compatibility with role attributes. More information can be found on this on the AWS OpsWorks User Guide.

For our Chef NFS cookbook, we need to know the requirements for a standard NFS server:

1. rpcbind enabled and running
2. NFS daemon (nfs service) enabled and running.

Chef typically uses Ruby for its file formats. Let's first create the directory structure:

```
$ mkdir opsworks
$ cd opsworks
$ mkdir alanscookbook
$ cd alanscookbook
$ cat > metadata.rb << __END__
name                  "alanscookbook"
maintainer            "Alan Landucci-Ruiz"
maintainer_email  "landucci@something-magical.net"
license               "GNU"
description           "Configures and starts Alans OpsWorks Stacks"
version               "1.0.0"
__END__
$ mkdir recipes
```

For AWS, this is the least amount that is needed, with the exception of the recipes. All recipe books go into the top-level of our git repository, which, in this case is the "opsworks" directory. Now, we need to create our NFS recipe.