Managing Multi-Tiered Applications with AWS OpsWorks

Daniele Stroppa

January 2015

Contents

Contents 2
Abstract 2
Introduction 3
Key Concepts 3
Design 5
Micro-services Architecture 5
Provisioning and Deployment 6
Managing Multiple Environments with AWS CloudFormation Templates 6
Continuous Integration, Continuous Delivery, Deployment Pipelines, and Continuous Deployment 7
Zero Downtime Deployments 8
Blue-Green Deployments 9
Monitoring 10
Amazon CloudWatch 10
Ganglia Built-in Layer and Custom Layers 12
Security 13
Amazon Virtual Private Cloud 13
Managing Access to the Instances 14
Managing Secrets 15
Conclusion 19
Further Reading 19

Abstract

An application usually requires a set of resources—such as a load balancer, web and application servers, and a database server—that you must create and manage as a whole. Additionally, you must deploy your application to the application servers, manage security and resource permissions, and monitor the performance of the solution.

This paper details how to use AWS OpsWorks to manage applications and their related resources and offers guidance to help deliver these solutions.

Introduction

AWS OpsWorks provides a flexible way to create and manage resources for your applications. It supports standard components such as application servers, database servers, and load balancers, as well as custom components such as search tools and messaging systems. AWS OpsWorks also provides tools to customize the standard package configurations, install additional packages, automate runbooks, and manage users' OS-level permissions.

Key Concepts

Before you get started using AWS OpsWorks, it's helpful to understand some key concepts.

Stack

A *stack* is a set of AWS resources that are managed together—for example, Amazon EC2 instances, Amazon EBS volumes, and Elastic Load Balancing load balancers. AWS OpsWorks helps you manage these resources as a whole and also defines some default configuration settings. You can create separate stacks for different environments—e.g., one stack for QA/testing and one for production—and for different applications.

Layer

Each stack contains one or more *layers*. A layer specifies how to configure a set of Amazon EC2 instances for a specific purpose such as hosting a web server.

AWS OpsWorks provides a set of built-in layers that support a variety of standard packages, including application servers such as Tomcat and Node.js, MySQL and Amazon RDS database servers, Elastic Load Balancing and HAProxy load balancers, and so on. AWS OpsWorks allows you to customize or extend the built-in layers by modifying default configurations and adding custom Chef recipes. You can also create a fully custom layer, which gives you complete control over the layer configuration and setup.

App

You represent your application within AWS OpsWorks by defining an *app*, which specifies the application type and contains the information needed to deploy the application from its repository to your application server instances. When deploying an app, AWS OpsWorks runs the Deploy recipes on all of the stack's instances, which allows instances such as database servers to modify their configuration as appropriate. AWS OpsWorks supports the ability to deploy multiple apps per stack and per layer.

Life Cycle Events

Each layer within an AWS OpsWorks stack has a set of *life cycle* events that correspond to different stages in an instance's life cycle, such as setup or app deployment. Each life cycle event has an associated set of Chef recipes that are executed on each of the layer's instances to perform the required tasks. AWS OpsWorks provides built-in recipes to perform basic management, and you can add custom recipes to any life cycle event to script any configuration change that your application needs.

Setup

Once a new instance has booted, AWS OpsWorks triggers the Setup event, which runs recipes to set up the instance according to the layer configuration. For example, if the instance is part of the PHP App Server layer, the Setup recipes install the Apache and PHP packages. Once setup is complete, AWS OpsWorks triggers a Deploy event, which runs recipes to deploy your application to the new instance.

Configure

Whenever an instance enters or leaves the online state, AWS OpsWorks triggers a Configure event on all instances in the stack. The event runs each layer's configure recipes to update the configuration to reflect the current set of online instances. For example, the HAProxy layer's Configure recipes modify the load balancer configuration to reflect any added or removed application server instances.

Deploy

AWS OpsWorks triggers a Deploy event when you run a Deploy command, typically to deploy your application to a set of application servers. The event runs recipes on the application servers to deploy the application and any related files from its repository to the layer's instances. You can trigger Deploy on other instances so they can, for example, update their configuration to accommodate the newly deployed app.

Undeploy

AWS OpsWorks triggers an Undeploy event when you delete an app or run an Undeploy command to remove an app from a set of application servers. The event runs recipes to remove all application versions and perform any additional cleanup tasks.

Shutdown

AWS OpsWorks triggers a Shutdown event when an instance is being shut down, but before the underlying Amazon EC2 instance is actually terminated. The event runs recipes to perform cleanup tasks such as shutting down services. AWS OpsWorks allows Shutdown recipes a configurable amount of time to perform their tasks, and then terminates the instance.

Design

An application is typically built using multiple tiers:

- Presentation – typically a front-end web server that serves static content, and potentially some cached dynamic content.
- Business logic – an application server where dynamic content is processed and generated.
- Workers – a set of servers that run background and long-running tasks.
- Data – usually a back-end database or data store.
- Integration – typically services to link the other layers together, e.g., messaging queues and topics.

AWS OpsWorks lets you model each tier with a layer that defines how to configure the tier's resources. You can also associate multiple layers with a single instance, for example if you need to configure an administrative web server for a group of web servers.

Micro-services Architecture

The micro-services architecture is a design approach to build a single application as a set of small services. Each service runs in its own process and communicates with other services via a well-defined interface using a lightweight mechanism, typically HTTP-based application programming interface (API). Micro-services are built around business capabilities, and each service performs a single function. Micro-services can be written using different frameworks or programming languages, and you can deploy them independently, as a single service, or as a group of services.

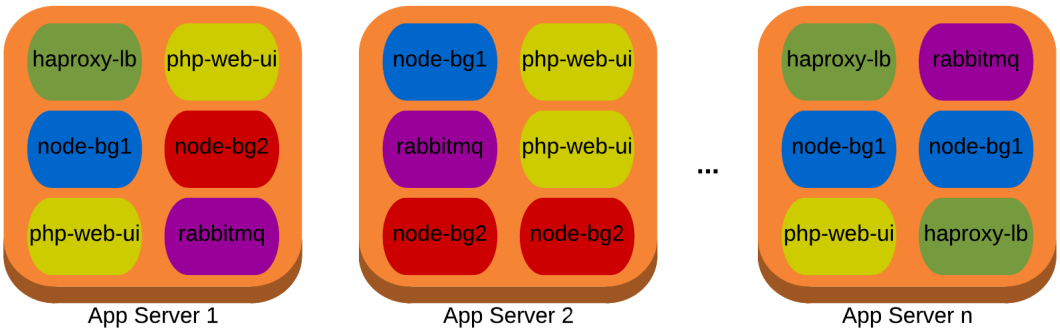

Figure 1: An example of an application architecture using micro-services

Micro-services can be modeled in AWS OpsWorks, with each micro-service being represented by a separate layer. Each layer can be configured independently using different recipes and each layer holds a set of instances. Application deployment can be done to a single instance, to all the instances in a particular layer – i.e., per micro-service – or to the instances in multiple layers at the same time.

Another approach to micro-service architectures is to use containers, such as Docker. Docker is an open-source platform that lets you build, deliver, and deploy distributed applications in lightweight execution environments called containers. These containers can scale according to business needs.

Docker's default container, libcontainer, enables image management and deployment services by using Linux kernel features such as:

- cgroups for resource isolation
- Namespaces to isolate the application's view of the operating environment
- Apparmor profiles to restrict the application's capabilities
- Networking interfaces to provide network connectivity
- Firewall rules to provide an isolated environment for applications.

Containers can share the same kernel, but each container can be limited to use only a specified amount of resources such as CPU, memory, and I/O.

Docker enables developers to create distributed systems easily by running multiple applications independently on a single machine. New resources are deployed as needed, enabling each micro-service to scale autonomously. The AWS Application Management Blog shows how to run your Docker containers with AWS OpsWorks.

Provisioning and Deployment

You should manage automated infrastructure provisioning like any other source code, by using a version control system. Provisioning resources, configuring software, and deploying applications should be deterministic, repeatable, flexible, and predictable. AWS OpsWorks, together with other AWS services such as AWS CloudFormation, enables you to provision the infrastructure needed to run your application and to manage continuous deployment of your application.

Managing Multiple Environments with AWS CloudFormation Templates

You can use AWS CloudFormation templates to model your AWS OpsWorks components—stacks, layers, instances, and applications—and provision them as AWS

CloudFormation stacks. This gives you the ability to track changes in your infrastructure using a version control tool and to share your AWS OpsWorks configuration. Additionally, you have the flexibility to create related services and resources, such as Elastic Load Balancing and Amazon RDS databases, alongside your AWS OpsWorks components using a single AWS CloudFormation template or nested AWS CloudFormation templates. These sample templates show how to model your AWS OpsWorks stack using AWS CloudFormation.

Using an AWS CloudFormation template, you'll be able to create multiple identical stacks, one for each deployment environment, e.g., test and production. In this way, you can make sure your application is being tested in an environment that is identical to the production environment. You can also define parameters in your AWS CloudFormation template, so that you'll be able to customize and differentiate stacks for different environments.

Continuous Integration, Continuous Delivery, Deployment Pipelines, and Continuous Deployment

With continuous integration, members of a development team integrate their work frequently, typically on a daily basis. Each integration is built and then tested to detect errors as quickly as possible.

With continuous delivery, you build software so that it can be released to production at any time. The code is deployable throughout its life cycle, which gives you fast and automated feedback on your systems' production readiness. You achieve continuous delivery by continuously integrating your code, building packages, and running automated tests to detect problems. You then deploy those packages into an environment as close as possible to your production environment using a deployment pipeline.

A deployment pipeline allows you to break up your build process into stages, with each stage giving you increased confidence in your build. Usually the first stage of a deployment pipeline compiles the code and provides packages for later stages. The final stage deploys the packages to production. The transition between stages can be automatic or require human authorization. You can usually detect build problems during the early stages of your pipeline, providing faster feedback and allowing you to terminate the process at that point. During later stages, troubleshooting requires thorough analysis and greater time.

You can also treat your pipeline as a single transaction: once a build is started, it's either successful and deployed to a production environment, or all changes in every stage are rolled back. This allows you to keep all your environments consistent at all times.

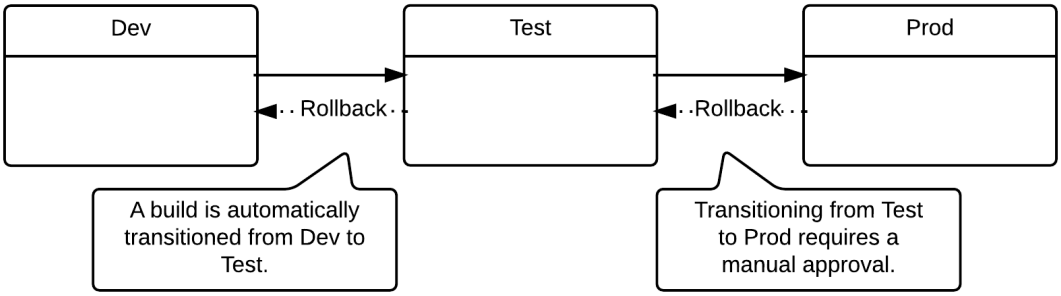

Figure 2 – An example of a deployment pipeline

Continuous deployment builds on continuous delivery concepts. With continuous delivery, the pipeline doesn't necessarily deploy code to production; you choose whether to deploy. With continuous deployment, every change that successfully goes through the pipeline is automatically deployed to production, typically resulting in multiple deployments per day.

Throughout the stages of your deployment pipeline, you'll need to deploy your code multiple times to different environments. There are different deployment methods:

- Bake a custom image, such as an Amazon Machine Images (AMI) or Docker container, that includes all required packages and configurations. Using baked images, you can provide consistent images for your environments, and you will allow for a quicker instance boot time. You can automate the baking process using tools such as Netflix Aminator, to create custom AMIs, or Packer, which also allows you to bake Docker images. Here you can find an example of a Packer template that you can use to bake an AMI.
- Deploy in place, meaning that you start your instances using a base AMI and then install all the packages and make the required configuration using Chef recipes. This can slow down the instance boot time, but it gives you more flexibility when configuring your environments.

You can choose to implement either of these methods or use a combination of both: bake a custom AMI with the packages and configurations that change less frequently and use Chef recipes to configure the more dynamic components.

Zero Downtime Deployments

When you are deploying multiple times a day to your production environment, you want to do so without causing any downtime for your application. During the AWS re:Invent 2014 session "Scalable Site Management Using AWS OpsWorks" we introduced a script that simplifies and automates zero downtime deployments with AWS OpsWorks.

The script uses the AWS SDK for Python to make calls to the AWS OpsWorks API. Upon execution, the script loops through the instances in an AWS OpsWorks layer and issues the deployment command to each one of them. If the layer is associated with an Elastic Load Balancing load balancer, the script will first deregister the instance from the load balancer, wait for the connection draining timeout setting, and then initiate the deployment. Once the deployment is complete, it will re-register the instance with the Elastic Load Balancing load balancer. The script will also read the load balancer health check configuration and wait for the instance to be considered healthy—i.e., healthy threshold times health check interval—before putting it back online.

Blue-Green Deployments

Blue-Green deployments are an efficient way for reducing deployment risk. The existing live production environment is referred to as *blue*, while the environment used to test a new version of your application is referred to as *green*. Once the new version has been successfully tested and is ready to go live, you can switch all user traffic from the blue environment to the green environment. The blue environment can be left idle and removed after a certain amount of time if no rollback is required.

You can implement blue-green deployments using AWS OpsWorks in combination with a pool of Elastic Load Balancing load balancers and Amazon Route 53.

- Both blue and green environments are represented by AWS OpsWorks stacks. The blue environment is running the current version while the green environment is running the new version. In combination with AWS CloudFormation each stack can have changes to software and resource configuration. Initially instances receive traffic only in the blue stack.
- Create a pool of Elastic Load Balancing load balancers that can be dynamically attached to a layer in either stack and that can be pre-warmed to cope with the expected volume of traffic.
- Use the weighted routing feature provided by Amazon Route 53 to create a record set in a hosted zone that includes all your pooled load balancers. Assign a zero weight to all load balancers not being used, and assign a nonzero weight to the load balancer attached to your live environment.

When you are ready for a blue-green switch:

- Create a green AWS OpsWorks stack identical to the existing blue stack; you can clone the blue stack or use an AWS CloudFormation template. Once the green stack is created, start instances in the required layers and deploy the new version of your application.
- Attach an Elastic Load Balancing load balancer from the pool to the application layer in the green stack.

- Once the new instances appear as healthy in the Elastic Load Balancing load balancer, change the weights in the Amazon Route 53 record set so that the load balancer attached to the green environment gets a nonzero weight, while the load balancer attached to the blue environment gets a zero weight.
- Detach the Elastic Load Balancing load balancer from the application layer in the blue stack so it can go back in the pool.
- Once there's no more need to keep the blue stack—i.e., once a rollback is not required—remove the stack.

Monitoring

Monitoring your resources is considered a best practice and enables organizations to identify and resolve infrastructure and application problems before they affect critical business processes.

AWS OpsWorks allows you to monitor your stacks and applications via Amazon CloudWatch—which provides metrics such as load, CPU, and memory—or with third-party tools such as Ganglia or Nagios.

Amazon CloudWatch

AWS OpsWorks uses Amazon CloudWatch to provide custom metrics with detailed monitoring for each instance in the stack and presents the aggregate data in each stack's monitoring section. You can view metrics for the entire stack or for a particular layer, or you can drill down to a specific instance.

You can also define custom metrics and publish them to Amazon CloudWatch with the put-metric-data command. Amazon CloudWatch stores metric data as a series of data points, each with an associated time stamp. You can publish one or more data points with each call to `put-metric-data`, and you can also publish an aggregated set of data points, known as a statistics set. Once your metrics have been published to Amazon CloudWatch, you can view the graphs of your metric in the AWS Management Console.

To publish your custom metric to Amazon CloudWatch from an AWS OpsWorks stack, you would typically write a custom recipe to create a cron job to publish your custom metrics using the AWS Command Line Interface (CLI) commands, the Amazon CloudWatch monitoring scripts for Linux or any of the available AWS SDKs. The following example shows a custom recipe that publishes disk space custom metrics using the Amazon CloudWatch monitoring scripts. The cron resource creates a cron job to run the script every five minutes:

```
# Install Perl dependecies
```

```
#
if platform?("ubuntu")
  package "unzip"
  package "libwww-perl"
  package "libcrypt-ssleay-perl"
  package "libswitch-perl"
elsif platform?("amazon")
  package "perl-Switch"
  package "perl-Sys-Syslog"
  package "perl-LWP-Protocol-https"
end

# Download the Amazon CloudWatch Monitoring Scripts for
Linux
#
remote_file "/opt/CloudWatchMonitoringScripts-v1.1.0.zip"
do
  source "http://ec2-downloads.s3.amazonaws.com/cloudwatch-
samples/CloudWatchMonitoringScripts-v1.1.0.zip"
  mode '0644'
end

# Unzip the Amazon CloudWatch Monitoring Scripts for Linux
#
execute "unzip" do
  command "unzip CloudWatchMonitoringScripts-v1.1.0.zip"
  creates "/opt/aws-scripts-mon"
  cwd "/opt"
end

# Add the script to cron so that it runs every 5 minutes
#
cron "cloudwatch-disk-space" do
  hour "*"
  minute "*/5"
  weekday "*"
  command "/opt/aws-scripts-mon/mon-put-instance-data.pl --
disk-space-used --disk-space-avail --disk-space-util --
disk-path=/ --from-cron"
end
```

Note that the Amazon EC2 instances in the AWS OpsWorks stack will need to have an AWS Identity and Access Management (IAM) role with a policy that allows the `cloudwatch:PutMetric*` actions.

Use Amazon CloudWatch to Monitor Stack Logs

Each instance in your stack will produce different log files, such as system logs, application logs, and perhaps custom logs. You probably would want to monitor some of

these logs to detect unexpected behavior or errors—e.g., when an application server is generating more 404 HTTP status codes than expected. AWS OpsWorks supports Amazon CloudWatch Logs to enable monitoring selected logs on multiple instances,

With Amazon CloudWatch Logs, you can monitor a log for the occurrence of a specific pattern. For example, you can monitor your application logs for the occurrence of a literal term such as ERROR. The Amazon CloudWatch Logs agent will send the logs to Amazon CloudWatch Logs, and you can then use the Amazon CloudWatch Logs console or the CLI to configure metrics to send you a notification when a certain condition is met, e.g., when the number of errors in the log exceeds a specified threshold.

To enable Amazon CloudWatch Logs monitoring on your AWS OpsWorks stack, you must follow these steps:

- Update your instance profile so that the Amazon CloudWatch Logs agent has the right permissions. You can use the same updated profile for all your instances.
- Create a configuration file that specifies details such as which logs to monitor, and install it in each instance's /tmp directory.
- Install and start the Amazon CloudWatch Logs agent on each instance.

You can automate the last two steps using custom recipes to handle the required tasks and assigning them to the appropriate layer's Setup events. Each time you start a new instance on those layers, AWS OpsWorks automatically runs your recipes after the instance finishes booting, enabling Amazon CloudWatch Logs. See the Quick Start: Install the CloudWatch Logs Agent Using AWS OpsWorks and Chef section in the AWS OpsWorks documentation for more information.

Ganglia Built-in Layer and Custom Layers

In addition to using Amazon CloudWatch to monitor the resources in your stack, you can also use the AWS OpsWorks Ganglia layer for additional application monitoring. The Ganglia layer is a blueprint for a Ganglia master instance that monitors your stack by using Ganglia distributed monitoring.

Typically, a stack includes only one Ganglia master instance. The standard AWS OpsWorks recipes install a low-overhead Ganglia client on every instance. If your stack includes a Ganglia layer, the Ganglia client will automatically report to the Ganglia master once the instance comes online. The Ganglia master uses these data to compute different statistics and displays the results via a web graphical interface.

The Chef Community also provides cookbooks for other popular monitoring tools such as Nagios, Monit, and Munin. You can easily add one of these tools in your stack using a custom layer.

Security

Security is a core functional requirement that protects mission-critical information from accidental or deliberate mishandling such as theft, leakage, and deletion.

AWS provides a secure global infrastructure and foundation services—compute, storage, networking and database—as well as higher level services. AWS provides a range of security services and features that customers can use to secure their assets. Under the shared responsibility model, AWS customers are responsible for protecting their data in the cloud and for meeting specific information protection requirements.

AWS OpsWorks can leverage the standard AWS security features to meet customers' security requirements and to provide a secure deployment environment.

Amazon Virtual Private Cloud

Amazon Virtual Private Cloud (Amazon VPC) enables you to encapsulate resources into a virtual network that you define and that is logically isolated from other virtual networks in the AWS cloud.

An Amazon VPC includes one or more subnets, each with an associated routing table that directs traffic based on its destination IP address.

- Instances within an Amazon VPC can communicate with each other regardless of their subnet.
- Subnets whose instances can communicate with the Internet are referred to as *public subnets.* They communicate with the Internet via an Internet Gateway
- Subnets whose instances cannot communicate directly with the Internet are referred to as *private subnets*. They can communicate with other instances in the Amazon VPC, but must communicate with the Internet via a network address translation (NAT) instance.

AWS OpsWorks requires that an Amazon VPC be configured so that all instances in the stack, including those in private subnets, have access to the AWS OpsWorks and Amazon S3 endpoints. You will also need access to the package repository for your operating system and other dependencies such as Ruby gems. For instances in private subnets, you can use a NAT to provide outbound connectivity to the Internet. See the Running a Stack in a VPC section in the documentation for more information.

Amazon VPC provides two features that can be used to increase security for your resources:

- Security groups act as a virtual firewall for your instance to control inbound and outbound traffic.

- Network access control lists (ACLs) act as a firewall for associated subnets, controlling both inbound and outbound traffic at the subnet level.

You can associate one or more security groups with your instances, and each instance in your Amazon VPC could belong to a different set of security groups. Additionally, you can further secure your instances by adding network ACLs to the subnets in your Amazon VPC.

Managing Access to the Instances

Directly accessing your instances via SSH or RDP is generally not a best practice and should be avoided. However, for troubleshooting purposes and when the information in the logs in not enough, you may need to login to your instances. Using key pairs is the default (and preferred) option to access your instances as it reduces the chance of somebody gaining access to the instance by guessing the password.

The “Managing OS-level Access to Amazon EC2 Instances” section in the AWS Security Best Practices whitepaper details how key pairs can be generated and how they fit in the instance booting process.

Use AWS OpsWorks Permissions to Set Users’ Public SSH Keys

AWS OpsWorks allows you to select IAM users for each stack and define each user’s permissions by using the AWS OpsWorks Permissions page or by attaching an appropriate IAM policy. Using the AWS OpsWorks permissions page, you can control which users have SSH access and sudo privileges on each instance in an AWS OpsWorks stack. Each IAM user can register a public SSH key with AWS OpsWorks. Once a public key is registered for an IAM user, you can grant privileges on a per-stack basis to use that key to connect to the stack’s instances. For each such instance, AWS OpsWorks will create an OS user, place the public key in the authorized_keys file, and update the public key if the user changes it.

Bastion Host

Using key pairs with a bastion host can be challenging. In particular, it’s best to avoid using the bastion host to store private keys that are required to connect to the instance. One way to overcome this issue is to use SSH agent forwarding on the client. This enables you to connect from the bastion host to your instances without the need to store the private key on the bastion host. The AWS Security Blog details how to configure and use SSH agent forwarding to connect to your instances.

When adding new instances to your stacks, you need to keep the hosts file on the bastion host up-to-date. To facilitate this process, you can use this cookbook, which creates a cron job to periodically update the host file.

Also, remember the following best practices when configuring your bastion host:

- When configuring the security group on the bastion host, apply the principle of least privilege, allowing SSH connections—i.e., port TCP/22—only from known and trusted IP addresses, such as your corporate network.
- You should have a bastion in each Availability Zone where your instances are. If your deployment takes advantage of an Amazon VPC virtual private network (VPN) connection, also have a bastion on premises.
- Configure the instances in your Amazon VPC to accept SSH connections only from a bastion host instance.
- Use the bastion host instance only as a bastion host and not for anything else. Additionally, you can harden the instance security further, e.g., enable SELinux, use a remote syslog server for logs, and configure host-based intrusion detection.

Managing Secrets

Secrets—such as database passwords, AWS credentials or third-party API credentials—must be stored and made available to servers that need to use them. If a server doesn't require a specific secret, it should not have access to it.

AWS OpsWorks Custom JSON or Chef data bags can be used to make data available to nodes in an AWS OpsWorks stack. However, these methods are not suitable to share secrets, as they are available in the clear to all stack users.

AWS OpsWorks Environment Variables

With AWS OpsWorks, you can define up to 20 environment variables for each application. These variables are passed to the application server instances during instance setup and can be updated on each application deployment. Custom layers can use a recipe to retrieve a variable's value using standard Chef node syntax and then store it in a form that is accessible to the layer's instances.

You can also define environment variables as protected values, so that they cannot be read by AWS OpsWorks users. For example, you can set separate environment variables for your database username and password. The password environment variable can be defined as a protected value, so it cannot be viewed in the console, AWS SDK, or CLI and is made available only to a specific application through JSON passed to specific instances for use in Chef recipes.

Encrypted Data Bags with a Secret File on Amazon S3 with Server-side Encryption (SSE)

Encrypted data bags use a shared secret and symmetric encryption of the data bag values. Data is encrypted using AES-256-CBC using a random initialization vector each time a value is encrypted to help protect against some forms of cryptanalysis. Only the values of a data bag item are decrypted, while keys are still searchable. Encrypted data bags can be decrypted only by a node or a user with the same shared secret.

This example shows how to modify the RDS-backed Simple Application Server Stack from the AWS OpsWorks Getting Started guide to use an encrypted data bag to store the DB username and password and to keep the secret file on an encrypted Amazon S3 bucket. You will need to have the Chef Development Kit (ChefDK) installed to create the encrypted data bag using the `knife` command; follow these instructions to install ChefDK.

First, let's create the secret file that will be used to encrypt the data bag:

```
$ openssl rand -base64 512 | tr -d '\r\n' > secret_file
```

The resulting file can be uploaded to an Amazon S3 bucket, making sure to enable server-side encryption for the object.

Make sure the IAM role attached to the instances in your stack—the default role is `aws-opsworks-ec2-role`—includes a policy to allow access to the Amazon S3 bucket where you uploaded your secret file:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt112233445566",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::bucket_name/secret_file"
      ]
    }
  ]
}
```

Create a plain text JSON file, containing the secrets to be shared:

```
$ echo "{\"id\": \"rdscredentials\", \"user\":
\"username\", \"password\": \"Passw0rd\"}" >>
plain_text.json
```

Create and encrypt the data bag using the secret file created before:

```
$ knife data bag create rdscredentials –z
Created data_bag[rdscredentials]
$ knife data bag from file rdscredentials
/path/to/plain_text.json --secret-file /path/to/secret_file
–z
Updated data_bag_item[rdscredentials::rdscredentials]
```

Modify the Stack Custom JSON to include your encrypted data bag and the details of the secret key on Amazon S3.

```
{
  "deploy": {
    "simplephpapp": {
      "database": {
        "database": "my_rds_db",
        "host": "my-rds.xxxxxxxxxx.eu-west-
1.rds.amazonaws.com",
        "adapter": "mysql"
      }
    }
  },
  "opsworks": {
    "data_bags": {
      "rdscredentials": {
        "rdscredentials": {
          "id": "rdscredentials",
          "user": {
            "encrypted_data":
"zVrVESc4NDR9nHSQxY1YLL5aodcx+9r68J1wnJh2tEY=\n",
            "iv": "iDNa3Cr4X0GttXPtoFEOaQ==\n",
            "version": 1,
            "cipher": "aes-256-cbc"
          },
          "password": {
            "encrypted_data":
"cBYeshea0ps9JS4YBNsxt9VQEQTMkNF/q+6B4ZKWmS4=\n",
            "iv": "FtZGMPPOkoxKG9xQ3EF3xg==\n",
            "version": 1,
            "cipher": "aes-256-cbc"
          }
        }
      }
    }
  },
  "secret": {
    "bucket": "bucket_name",
    "object": "secret_file"
```

```
  }
}
```

Modify the appsetup.rb recipe so that the template resource reads the secret file from the Amazon S3 bucket and the values from the encrypted data bag are used to compile the template.

```
require 'rubygems'
require 'aws-sdk'

node[:deploy].each do |app_name, deploy|

  script "install_composer" do
    interpreter "bash"
    user "root"
    cwd "#{deploy[:deploy_to]}/current"
    code <<-EOH
    curl -sS https://getcomposer.org/installer | php
    php composer.phar install --no-dev
    EOH
  end

  template "#{deploy[:deploy_to]}/current/db-connect.php"
do
    source "db-connect.php.erb"
    mode 0660
    group deploy[:group]

    if platform?("ubuntu")
      owner "www-data"
    elsif platform?("amazon")
      owner "apache"
    end

    s3 = AWS::S3.new()
    secret =
s3.buckets[node[:secret][:bucket]].objects[node[:secret][:o
bject]].read.strip

    rdscredentials =
Chef::EncryptedDataBagItem.load("rdscredentials",
"rdscredentials", secret)

    variables(
      :host =>     (deploy[:database][:host] rescue nil),
      :user =>     (rdscredentials['user']),
      :password => (rdscredentials['password']),
```

```
      :db =>       (deploy[:database][:database] rescue
nil),
      :table =>    (node[:phpapp][:dbtable] rescue nil)
    )

   only_if do
     File.directory?("#{deploy[:deploy_to]}/current")
   end
  end
end
```

Conclusion

In this paper, we showed how you could use AWS OpsWorks to manage complex multi-tiered applications, from designing a scalable and flexible architecture to continuously provisioning and deploying infrastructure and applications. We also highlighted how monitoring and security play an important role in such deployments and how AWS OpsWorks enables you to easily manage these aspects.

Further Reading

For more information about managing multi-tiered applications with AWS OpsWorks, see the following sources.

- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
- Continuous Integration and Deployment Best Practices on AWS
- Scalable Site Management Using AWS OpsWorks
- AWS Security Best Practices